Safety Verification of Phaser Programs

Zeinab Ganjei, Ahmed Rezine, Petru Eles, Zebo Peng
dept. of computer and information science
Linköping University, Sweden
firstname.surname@liu.se

***Abstract*—We address the problem of statically checking control state reachability (as in possibility of assertion violations, race conditions or runtime errors) and plain reachability (as in deadlock-freedom) of *phaser programs*. Phasers are a modern non-trivial synchronization construct that supports dynamic parallelism with runtime registration and deregistration of spawned tasks. They allow for collective and point-to-point synchronizations. For instance, phasers can enforce barriers or producer-consumer synchronization schemes among all or subsets of the running tasks. Implementations are found in modern languages such as Habanero Java. Phasers essentially associate phases to individual tasks and use their runtime values to restrict possible concurrent executions. Unbounded phases may result in infinite transition systems even in the case of programs only creating finite numbers of tasks and phasers. We introduce an exact gap-order based procedure that always terminates when checking control reachability for programs generating bounded numbers of coexisting tasks and phasers. We also show verifying plain reachability is undecidable even for programs generating few tasks and phasers. We then explain how to turn our procedure into a sound analysis for checking plain reachability (including deadlock freedom). We report on preliminary experiments with our open source tool.**

***Index Terms*—phasers, safety verification, dynamic synchronization, collective synchronization, Point-to-point synchronization, model checking**

I. Introduction

We focus on safety verification of programs using *phasers* for task synchronization [1]–[3]. This sophisticated construct dynamically unifies collective and point-to-point synchronizations. For instance, it allows for dynamic registration and deregistration of tasks allowing for a more balanced usage of the computing resources when compared to static producer-consumer or barrier constructs [4]. The construct can be added to any parallel programming language with a shared address space. For instance, it can be found in Habanero Java [3], an extension of the Java programming language. Phasers build on the clock construct from the X10 programming language [1]. They can be created dynamically and spawned tasks may get registered or deregistred at runtime.

Intuitively, each phaser associates two phases (hereafter *wait* and *signal* phases) to each registered task. Apart from creating phasers and registering each other to them, tasks can individually issue `wait` and `signal` commands to a phaser they are registered to. Intuitively, `signal` commands are used to inform other registered tasks the issuing task is done with its *signal* phase. The command is non-blocking. It increments the *signal* phase associated to the issuing task on the given phaser. The `wait` command is instead used to check whether all registered tasks are done with (i.e., have a *signal* phase that is strictly larger than) the issuing task's *wait* phase. This command may get blocked by a task that did not yet finish the corresponding phase. Unlike classical barriers, phasers need not force registered tasks to wait for each other at each single phase. Instead they allow them to proceed with the following phases (by issuing `signal` commands), or even to exit the construct by deregistering from the phaser. Such dynamic behavior allows for better load balancing and performance, but comes at the price of making it easy to introduce programming mistakes such as assertion violations, race conditions, runtime errors and, in the important situation where wait and signal commands are decoupled for maximum flexibility, deadlocks. We summarize our contributions in this work:

- We propose an operational model based on [2], [3], [5].
- We show undecidability of checking deadlock-freedom for programs with fixed numbers of tasks and phasers.
- We describe an exact gap-order based symbolic verification procedure for checking control state reachability (as in assertion violations, race conditions or runtime errors) and plain reachability (as in checking deadlock freedom).
- We show termination of the procedure for control state reachability when numbers of tasks and phasers are fixed.
- We describe how to turn the procedure into a sound over-approximation for plain reachability.
- We report on our preliminary experiments with our open source tool.

Related work. We are not aware of automatic formal verification works that focus on constructs allowing for such a degree of dynamic parallelism. Unlike [6], we focus on fully automatic verification and consider the richer and more challenging phaser construct. The work of [5] considers the dynamic verification of phaser programs and can therefore only reason about particular program inputs and runs. The work in [7] uses Java Path Finder [8] to explore several runs, but still for one concrete input at a time. The works in [9], [10] target gap-order systems. Although phaser programs share some of their properties (larger gaps can do more), the results in [9], [10] do not apply since gap-order systems crucially forbid exact increments.

Outline. We describe a phaser program and recall some preliminaries in Sections II and III. This is followed in Section IV by a formal description of phaser programs and

This work is partially supported by the CENIIT research organization.

of the properties we want to check. We also establish the undecidability of checking deadlock freedom. We introduce a gap-order based symbolic representation in Section V and describe in Section VI a simple verification procedure. We then show decidability of checking control state reachability and introduce a relaxation procedure for checking plain reachability. Finally, we report on our experiments and conclude the work. Descriptions of the proofs can be found in [11].

II. Motivating example

The program listed in Fig. (1) uses Boolean shared variables $\mathtt{B} = \{\mathtt{a}, \mathtt{b}, \mathtt{done}\}$. A *main* task creates two phasers (lines 5 and 6). When creating a phaser, the task gets automatically registered to it. The main task also creates three other task instances (lines 9, 10 and 11). Several tasks can be registered to several phasers. When a task t is registered to a phaser p, a pair of numbers $(wait_p^t, sig_p^t)$, each in $\mathbb{N} \cup \{+\infty\}$, is associated to the couple (t, p). The pair represents the individual *wait* and *signal* phases of task t on phaser p.

Registration of a task to a phaser can occur in one of three modes: SIG_WAIT, WAIT and SIG. In SIG_WAIT mode, a task may issue both `signal` and `wait` commands. In WAIT mode, a task may only issue `wait` commands on the phaser. Finally, when registered in SIG mode, a task may only issue `signal` commands. Issuing a `signal` command by a task on a phaser results in the task incrementing its signal phase associated to the phaser. This command is non-blocking. On the other-hand, issuing a `wait` command by a task on a phaser p will block until **all** tasks registered on p exhibit signal values on p that are strictly larger than the wait value of the issuing task on phaser p. In this case, the wait phase of the issuing task is incremented. Intuitively, a signal command allows the issuing task to state other tasks need not wait for it to complete its signal phase. In retrospect, a `wait` command allows a task to make sure all registered tasks have moved past its wait phase.

Upon creation of a phaser, wait and signal phases are initialized to 0 (except in WAIT mode where the signal phase is instead initialized to $+\infty$ in order to not block other waiters). The only other way a task may get registered to a phaser is if an already registred task does register it in the same mode (or in WAIT or SIG if the registrar is registered in SIG_WAIT). In this case, wait and signal phases of the newly registered task are initialized to those of the registrar. Tasks are therefore dynamically registered (e.g., lines 9-11). They can also dynamically deregister themselves (e.g., lines 25-26);

In this example, two producers and one consumer are synchronized using two phasers. The consumer requires the two producers to be ahead of it (wrt. the phaser main pointed to with `prod`) in order for it to consume their respective products. At the same time, the consumer needs to be ahead of both producers (wrt. the phaser main pointed to with `cons`) in order for these to produce their pair of products. It should be clear that phasers can be used as barriers for synchronizing dynamic subsets of concurrent tasks. Observe producers need not, in general, proceed in a lock step fashion. Producers may produce many items before consumers "catch up".

We are interested in checking: (a) control state reachability as in assertions (e.g., line 44), race conditions (e.g., mutual exclusion of lines 20 and 49) or runtime errors (e.g., signaling a dropped phaser), and (b) plain reachability as in deadlocks (e.g., a producer at line 23 and a consumer at line 50 with equal phases). Intuitively, both problems concern themselves with the reachability of target sets of program configurations. The difference is that control state reachability defines the targets with the states of the tasks (their control locations and whether they are registered to some phasers). Plain reachability can, in addition, use values or relations between values of involved phases. Observe that control state reachability depends on the values of the actual phases, but these values are not used to define the target sets. For example, assertions are expressed as predicates over Boolean variables (e.g., line 44). Establishing such an assertion requires capturing the constraints imposed by the phasers on the program behaviors.

Our work proposes a sound and complete algorithm for checking control state reachability in case a bounded number of tasks and phasers are generated. The algorithm can handle arbitrarily large phases, e.g., generated using nested signaling loops. The algorithm starts from a symbolic representation of all bad configurations and successively computes sets of predecessor configurations. We show termination based on a well-quasi-ordering argument that imposes restrictions on what can be expressed with our symbolic representation. For instance putting upper bounds on differences between phases is forbidden. Deadlock configurations cannot be faithfully captured with such restricted representations. Intuitively, a deadlocked configuration will have a cycle where each involved task is waiting for the task to its right but where the wait phase of each task equals the signal phase of the task it is waiting for. We show the problem of checking deadlock freedom to be undecidable even for programs only generating a bounded number of tasks and phasers. We explain how to turn our verification algorithm into a sound but incomplete procedure for checking deadlock-freedom. Precision can then be augmented on demand to eliminate false positives.

III. Preliminaries

We use $\mathbb{N}$ and $\mathbb{Z}$ for natural and integer numbers respectively. We write $A \uplus B$ to mean the union of disjoint sets A and B. We let $\mathtt{Pfn}(A, B)$ be the set of partial functions from A to B and use $\varnothing_A$ for the empty function over A, i.e., $\varnothing_A(a)$ is undefined (written $\varnothing_A(a) \uparrow$) for all $a \in A$. Given function $g \in \mathtt{Pfn}(A, B)$ we write $g(a) \downarrow$ to mean that $g(a)$ is defined and write $g \setminus \{a\}$ to mean the restriction of g to the domain $A \setminus \{a\}$. We write $g[a \leftarrow b]$ for the function that coincides with g on A except for a that is sent to b. We abuse notation and let, for pairwise different $\{a_i \mid i \in I\}$, $g[\{a_i \leftarrow b_i \mid i \in I\}]$ mean the function that coincides with g on A except for each a_i that is sent to the corresponding b_i. We sometimes write a function g as a set $\{a \mapsto g(a) \mid a \in A\}$. It is then implicitly undefined outside of A.

```
1  bool a, b, done;
2  main()
3  {
4    done = false;
5    prod = newPhaser(SIG_WAIT);
6    cons = newPhaser(SIG_WAIT);
7    cons.signal();
8
9    asynch(aProducer, prod(SIG), cons(WAIT));
10   asynch(bProducer, prod(SIG), cons(WAIT));
11   asynch(abConsumer, prod(WAIT), cons(SIG));
12
13   prod.drop();
14   cons.drop();
15 }
16
17 aProducer(p(SIG), c(WAIT))
18 {
19   c.wait();
20   while(¬done){
21     a = true;
22     p.signal();
23     c.wait();
24   };
25   p.drop();
26   c.drop();
27 }
```

```
28  bProducer(p(SIG), c(WAIT))
29  {
30    c.wait();
31    while(¬done){
32      b = true;
33      p.signal();
34      c.wait();
35    };
36    p.drop();
37    c.drop();
38  }
39
40 abConsumer(p(WAIT), c(SIG))
41 {
42    while(¬done){
43     p.wait();
44     assert(a ∧ b);
45     a = false;
46     b = false;
47
48     if(ndet())
49      done = true;
50     c.signal();
51   };
52   c.drop();
53   p.drop();
54 }
```

Fig. 1. Two producers and one consumer are synchronized using two phasers. In this construction, the consumer requires both producers to be ahead of it (wrt. the `prod` phaser) in order for it to consume their respective products. At the same time, the consumer needs to be ahead of both producers (wrt. the `cons` phaser) in order for these to be able to produce their pair of products.

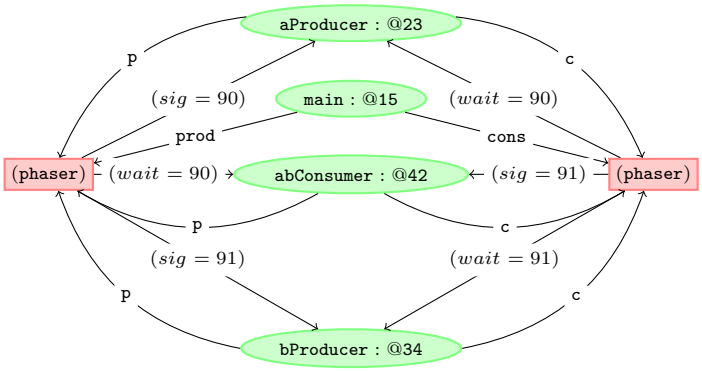

Fig. 2. Possible wait and signal phase values for Fig. (1). Observe that there is no a priori bound on the values of the different wait and signal phases. In this example, the difference between signal and wait phases is bounded. This is not always the case in general.

IV. Language

A program may use a set $\mathtt{B}$ of shared Boolean variables and a set $\mathtt{V}$ of local phaser variables:

$$
\begin{array}{lll}
\mathtt{prg} & ::= & \mathtt{bool}\ \mathtt{b}_1, \ldots, \mathtt{b}_{|\mathtt{B}|}; \\
 & & \mathtt{task}_1(\mathtt{v}_{1_1}, \ldots, \mathtt{v}_{k_1})\,\{\mathtt{stmt}_1\} \\
 & & \ldots \\
 & & \mathtt{task}_n(\mathtt{v}_{1_n}, \ldots, \mathtt{v}_{k_n})\,\{\mathtt{stmt}_n\} \\
 & & \\
\mathtt{stmt} & ::= & \mathtt{v} = \mathtt{newPhaser}() \mid \mathtt{asynch}(\mathtt{task}, \mathtt{v}_1, \ldots, \mathtt{v}_k) \\
 & & \mid \mathtt{v.drop}() \mid \mathtt{v.signal}() \mid \mathtt{v.wait}() \mid \mathtt{exit} \\
 & & \mid \mathtt{stmt}; \mathtt{stmt} \mid \mathtt{b} = \mathtt{cond} \mid \mathtt{assert}(\mathtt{cond}) \\
 & & \mid \mathtt{while}(\mathtt{cond})\,\{\mathtt{stmt}\} \mid \mathtt{if}(\mathtt{cond})\,\{\mathtt{stmt}\} \\
\mathtt{cond} & ::= & \mathtt{ndet}() \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{b} \mid \mathtt{cond} \vee \mathtt{cond} \\
 & & \mid \mathtt{cond} \wedge \mathtt{cond} \mid \neg\mathtt{cond}
\end{array}
$$

A program consists in a set of tasks $\mathtt{T}$. A task is declared with $\mathtt{task}(\mathtt{v}_1, \ldots, \mathtt{v}_k)\,\{\mathtt{stmt}\}$ where $\mathtt{v}_1, \ldots \mathtt{v}_k$ are phaser variables that are local to the declared task. A task can also create a new phaser with $\mathtt{v} = \mathtt{newPhaser}()$ and store the identifier of the phaser in a local variable $\mathtt{v}$. We let $\mathtt{V}$ be the union of all local phaser variables. When creating a phaser, a task gets registered to it. To simplify our description, we will assume all registrations to be in SIG_WAIT mode. Including the other modes is a matter of changing the initial phase values at registration and of statically ensuring the issued commands respect the registration mode. A task can deregister itself from a phaser referenced by a variable $\mathtt{v}$ with $\mathtt{v.drop}()$. It can also issue signal or wait commands on a phaser on which it is registered and that is referenced by $\mathtt{v}$. A task can spawn another task with $\mathtt{asynch}(\mathtt{task}, \mathtt{v}_1, \ldots, \mathtt{v}_n)$. The issuing task registers the spawned task to the phasers it points to with $\mathtt{v}_1, \ldots, \mathtt{v}_n$. The issuing task need not wait for the spawned task and may directly continue its execution.

Assume a phaser program $\mathtt{prg} = (\mathtt{B}, \mathtt{V}, \mathtt{T})$. We inductively define the finite set $\mathtt{S}$ of control sequences as follows. $\mathtt{S}$ is the smallest set containing: (i) suffixes of each "$\mathtt{stmt}_i$" appearing in some "$\mathtt{task}_i(\mathtt{v}_{1_i}, \ldots, \mathtt{v}_{k_i})\,\{\mathtt{stmt}_i\}$"; and (ii) suffixes of "$\mathtt{stmt}_i; \mathtt{while}(\mathtt{cond})\,\{\mathtt{stmt}_i\}; \mathtt{stmt}_j$" (respectively "$\mathtt{stmt}_i; \mathtt{while}(\mathtt{cond})\,\{\mathtt{stmt}_i\}$") for each "$\mathtt{while}(\mathtt{cond})\,\{\mathtt{stmt}_i\}; \mathtt{stmt}_i$" (respectively "$\mathtt{while}(\mathtt{cond})\,\{\mathtt{stmt}_i\}$") in $\mathtt{S}$; and (iii) suffixes of "$\mathtt{stmt}_i; \mathtt{stmt}_j$" (respectively "$\mathtt{stmt}_i$") for each "$\mathtt{if}(\mathtt{cond})\,\{\mathtt{stmt}_i\}; \mathtt{stmt}_j$" (respectively "$\mathtt{if}(\mathtt{cond})\,\{\mathtt{stmt}_i\}$") appearing in $\mathtt{S}$. We write $\mathtt{s}$ to mean some control sequence in $\mathtt{S}$, and $\mathtt{hd}(\mathtt{s})$ and $\mathtt{tl}(\mathtt{s})$ to respectively mean the head and the tail of the sequence $\mathtt{s}$.

A. Semantics.

A configuration c of $\texttt{prg} = (\texttt{B}, \texttt{V}, \texttt{T})$ is a tuple $(\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}, \boldsymbol{pv}, \boldsymbol{\varphi})$ where:

- $\mathcal{T}$ is the current finite set of task identifiers. We let t, u range over the values in $\mathcal{T}$.
- $\mathcal{P}$ is the current finite set of phaser identifiers. We let p, q range over the values in $\mathcal{P}$.
- $\boldsymbol{bv} : \texttt{B} \to \{\texttt{true}, \texttt{false}\}$ is a total mapping that associates a value to each $\texttt{b} \in \texttt{B}$.
- $\boldsymbol{pc} : \mathcal{T} \to \texttt{S}$ is a total mapping that associates tasks to their remaining sequences (i.e., control location).
- $\boldsymbol{pv} : \mathcal{T} \to \texttt{Pfn}(\texttt{V}, \mathcal{P})$ is a total mapping that associates, to each task identifier in $\mathcal{T}$, a partial mapping from the local phaser variables V to phaser identifiers $\mathcal{P}$. It captures the values of the phaser variables V of each task.
- $\boldsymbol{\varphi} : \mathcal{P} \to \texttt{Pfn}(\mathcal{T}, \mathbb{N}^2)$ is a total mapping that associates to each phaser $p \in \mathcal{P}$ a partial mapping $\boldsymbol{\varphi}(p)$ that is defined exactly on the identifiers of the tasks registered to p. For such a task t, $\boldsymbol{\varphi}(p)(t)$ is the pair $(wait_p^t, sig_p^t)$ representing wait and signal values of t on p.

The set of tasks $\mathcal{T}$ is altered by $\texttt{asynch}(\texttt{task}, \texttt{v}_1, \ldots, \texttt{v}_\texttt{n})$ and `exit` statements (rules (`asynch`) and (`exit`) in Fig.(3)). The set of phasers $\mathcal{P}$ is updated upon creation of new phasers (rule (`newPhaser`) in Fig.(3)). The mapping $\boldsymbol{pv}$ associates values to program phaser variables. Accessing variables with undefined values, or phasers to which the task is not currently registered, leads to runtime errors (rule (`runtime error`)). The total mapping $\boldsymbol{\varphi}$ captures states of phasers. It associates to each phaser identifier p in $\mathcal{P}$ a partial mapping $\boldsymbol{\varphi}(p)$. This partial mapping is defined for a task identifier $t \in \mathcal{T}$ (i.e., $\boldsymbol{\varphi}(p)(t) \downarrow$) iff the task t is registered to the phaser p. In this case, $\boldsymbol{\varphi}(p)$ gives the waiting phase $wait_p^t$ and the signaling phase sig_p^t of the task t on the phaser p. Initially, a unique "main" task $\texttt{t}_0$ starts executing its $\texttt{stmt}_{main}$ with no phasers. $\boldsymbol{\varphi}$ is the empty function with an empty domain $\varnothing_\emptyset$. After a task t executes a $\texttt{v} := \texttt{newPhaser}()$ statement (rule (`newPhaser`) in Fig.(3)), a new phaser p is associated to the variable v using $\boldsymbol{pv}$ and $\boldsymbol{\varphi}(p)$ becomes the partial function $\{t \mapsto (0,0)\}$. The initial configuration is $c_{init} = (\{\texttt{t}_0\}, \{\}, \boldsymbol{bv}_{\texttt{false}}, \{\texttt{t}_0 \mapsto \texttt{stmt}\}, \varnothing, \varnothing)$, where a "main" task with identifier $\texttt{t}_0$ and code `stmt` is the unique initial task. No phasers are present in the initial configuration, and all Boolean variables are mapped to `false`.

Given two configurations c and c' with $c = (\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}, \boldsymbol{pv}, \boldsymbol{\varphi})$, we write $c \xrightarrow{t} c'$ if there is a task $t \in \mathcal{T}$ such that one of the rules in Fig.(3) holds. We use $\xrightarrow{*}$ for the reflexive transitive closure of $\to$ and write $c \xrightarrow{*} c'$ to mean that c' is reachable from c. A configuration is said reachable if it is reachable from the initial configuration c_{init}.

1) Control-state reachability: Checking the possibility of assertion violations, of runtime errors and of race conditions amounts to checking reachability of configurations respectively in $\texttt{badConfs}^{(n,p)}_{\texttt{assert}}$, $\texttt{badConfs}^{(n,p)}_{\texttt{runtime}}$ and in $\texttt{badConfs}^{(n,p)}_{\texttt{race}}$ for some number of tasks n and number of phasers p. We introduce in Section V a complete procedure for checking reachability of such sets of configurations and show it to be sound for programs with fixed upper bounds on numbers of generated phasers and tasks.

2) Deadlocks as in plain reachability: We are also interested in checking the possibility of deadlocks. For this we need to define the notion of a blocked task. Assume in the following a configuration $c = (\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}, \boldsymbol{pv}, \boldsymbol{\varphi})$.

Definition 1 (Blocked)**.** *A task $t \in \mathcal{T}$ is blocked at phaser $p \in \mathcal{P}$ by task $u \in \mathcal{T}$ if $\texttt{hd}(\boldsymbol{pc}(\texttt{t})) = \texttt{v.wait}()$ with $\boldsymbol{pv}(t)(\texttt{v}) = p$ and $\boldsymbol{\varphi}(p)(t) = (wait_p^t, \_)$ when $\boldsymbol{\varphi}(p)(u) = (\_, sig_p^u)$ and $sig_p^u \leq wait_p^t$.*

Intuitively, a task t is blocked by a task u if it cannot finish its `wait` command on some phaser because it is waiting for task u that did not issue enough `signal` commands on the same phaser.

Definition 2 (Deadlock)**.** *$(\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}, \boldsymbol{pv}, \boldsymbol{\varphi})$ is a deadlock configuration if each task of a non empty subset $\mathcal{U} \subseteq \mathcal{T}$ is blocked by some task in $\mathcal{U}$.*

Theorem 1 (Deadlock-Freedom)**.** *It is undecidable in general, even for programs with only three phasers and four tasks, to check for deadlock-freedom.*

The idea of the proof is to encode the reachability problem of any given 3-counters reset-VAS (vector addition system with reset arcs) as the reachability problem of a configuration with a cycle involving three phasers and three tasks (in addition to the main task). Indeed, reachability of configuration $(s_F, 0, 0, 0)$ (three counters with zero values at some control location s_F) is undecidable for reset-VASs. The idea then is to spawn three tasks and as many phasers. The value of each counter is captured with the difference between the signal and the wait of a pair of tasks on one phaser. Resets are encoded by asking a task to drop a phaser and exit and spawning a new task. The encoding ensures that a deadlock is reached exactly when the vector addition system reaches configuration $(s_F, 0, 0, 0)$. (See [11] for more details.)

V. Symbolic verification of phaser programs

We briefly introduce gap-order constraints and use them to define a symbolic representation (hereafter constraints) that we use in Section VI for checking reachability.

A. Gap-order constraints and graphs [9], [10], [12], [13].

Gap-order constraints can be regarded as a particular case of the octagons or the *unit two variables per inequality* (utvpi) constraints. Assume in this section that x and y are integer variables and that k is an integer constant. We use X and Y to mean finite sets of integer variables. A valuation val is a total function $X \to \mathbb{Z}$. Valuations are implicitly extended to preserve constants (i.e. $val(k) = k$ for any $k \in \mathbb{Z}$). A *gap-order clause* δ over X is an inequality of the form $a - b \geq k$ where $a, b \in X \cup \{0\}$. A *gap-order constraint* Δ over X is a finite conjunction of gap-order clauses over the same set X. Observe that $(x = y + 2 \ \wedge \ y \leq 5)$ is essentially a gap-order constraint because it can be equivalently rewritten as

Fig. 3. Operational semantics of phaser statements.

the conjunction $(x - y \geq 2 \ \wedge \ y - x \geq -2 \ \wedge \ 0 - y \geq -5)$. Given a gap-order constraint Δ over X and a valuation $val : X \to \mathbb{Z}$, we write $val \models \Delta$ to mean that $val(a) - val(b) \geq k$ holds for each gap-order clause $\delta : a - b \geq k$ appearing in Δ. We let $Sat(\Delta)$ be the set $\{val : X \to \mathbb{Z} \mid val \models \Delta\}$.

A gap-order graph (or graph for short) $\wp$ over X is a graph (V, E) with vertices $V = X \cup \{0\}$ where edges in E are of the form $a \xrightarrow{k} b$ with $a, b \in V$ and weight k in $\mathbb{Z} \cup \{-\infty, +\infty\}$. We let $\texttt{varsOf}(\wp) = X$. Given a gap-constraint Δ over X, we can build the graph $\texttt{graphOf}\,(\Delta)$ with vertices $X \cup \{0\}$ and where E only contains a representative $a \xrightarrow{k} b$ edge for each clause $a - b \geq k$ appearing in Δ. A valuation $val : X \to \mathbb{Z}$ satisfies a graph $\wp = (V, E)$ (written $val \models \wp$) iff $val(a) - val(b) \geq k$ for each $a \xrightarrow{k} b \in E$. We let $Sat(\wp)$ be the set $\{val : X \to \mathbb{Z} \mid val \models \wp\}$. Clearly, $Sat(\texttt{graphOf}\,(\Delta)) = Sat(\Delta)$. The closure $\texttt{clo}\,(\wp)$ of a graph $\wp = (V, E)$ is the unique complete graph with the same vertices V and where $a \xrightarrow{k'} b$ is an edge of $\texttt{clo}\,(\wp)$ iff $k' \in \mathbb{Z} \cup \{-\infty, +\infty\}$ is the least upper bound of all weight-sums for any path in $\wp$ from a to b. Closure allows us to deduce $(0 - x \geq -7)$ from $(y - x \geq -2 \wedge 0 - y \geq -5)$. The result of the closure procedure is a special graph $\wp_{\texttt{false}}$ denoting the graph without any satisfying valuation each time a weight k=$+\infty$ is generated. The closure of a graph can be computed in polynomial time and we get $Sat(\texttt{clo}\,(\wp)) = Sat(\wp)$. We define the *degree* of a graph $\wp$ (written $\texttt{degreeOf}(\wp)$) to be 0 if no edge in $\texttt{clo}\,(\wp)$ has a negative weight apart from $-\infty$. Otherwise, $\texttt{degreeOf}(\wp)$ is the largest natural $k \in \mathbb{N}$ such that there is an edge in $\texttt{clo}\,(\wp)$ with weight $-k$. For instance, the degree of the graph resulting from $(x - y \geq 2 \wedge y - x \geq -4)$ is 4. We systematically close all manipulated graphs and write $\mathcal{G}(X)$ for the set of closed graphs over X. Given a graph $\wp$, we write $\wp[x/y]$ to mean the graph obtained by replacing the vertex x by the vertex y. We abuse notation and write $\wp[\{x_i/y_i \mid i \in I\}]$, for pairwise different x_i elements to mean the simultaneous application of the individual substitutions. For a set of variables Y, we write $\wp \ominus Y$ to mean the graph obtained by removing the variables in Y from the vertices of $\wp$. Given two closed graphs $\wp$ and $\wp'$ over the same X, we write $\wp \sqsubseteq_{\mathcal{G}} \wp'$ to mean that each directed edge in $\wp$ is labeled with a larger weight in $\wp'$. As a result, $Sat(\wp') \subseteq Sat(\wp)$. Finally, we write $\wp \oslash \wp'$ to mean the closure of the graph obtained with merging the two sets of vertices and edges. As a result, $Sat(\wp \oslash \wp') = Sat(\wp) \cap Sat(\wp')$.

B. Constraints as a symbolic representation.

A constraint ϕ is a tuple $(\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}, \boldsymbol{pv}, \boldsymbol{\gamma})$ where the only difference with the definition of a configuration $(\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}, \boldsymbol{pv}, \boldsymbol{\varphi})$ is the adoption of a gap-order constraint $\boldsymbol{\gamma}$ instead of $\boldsymbol{\varphi}$. More specifically, $\boldsymbol{\gamma} : \mathcal{P} \to \cup_{\mathcal{U} \subseteq \mathcal{T}} \mathcal{G}(\cup_{t \in \mathcal{U}} \{\omega_p^t, \sigma_p^t\})$ is a total mapping that associates a gap-order graph to each phaser $p \in \mathcal{P}$. Intuitively, we use variables ω_p^t and σ_p^t to constrain in graph $\boldsymbol{\gamma}(p)$ possible values of both wait ($wait_p^t$) and signal (sig_p^t) phases of each task t registered to phaser p. As a result, we can check if task t is registered to phaser p according to graph $\wp = \boldsymbol{\gamma}(p)$ by checking if $\{\omega_p^t, \sigma_p^t\} \subseteq \texttt{varsOf}(\wp)$. We will write $\texttt{Reg}(p, \wp)$ to mean the set of tasks $\{t \mid \{\omega_p^t, \sigma_p^t\} \subseteq \texttt{varsOf}(\wp)\}$. We also write $\texttt{isReg}(t, p, \wp)$ for the predicate $t \in \texttt{Reg}(p, \wp)$. Observe that the language semantics impose that, for each phaser p and for any pair t, u of tasks in $\texttt{Reg}(p, \wp)$, the predicate $0 \leq wait_p^t \leq sig_p^u$ is an invariant. For this reason, we always safely strengthen, in any obtained $\boldsymbol{\gamma}(p) = \wp$, weights k in $\sigma_p^t \xrightarrow{k} \omega_p^u$, $\sigma_p^t \xrightarrow{k} 0$ and $\omega_p^t \xrightarrow{k} 0$ with $max(k, 0)$. The following definition helps us characterize configurations for which our procedure terminates.

Definition 3 (degree and freeness of constraints)**.** *A constraint $(\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}, \boldsymbol{pv}, \boldsymbol{\gamma})$ has as degree the largest degree among all its graphs $\boldsymbol{\gamma}(p)$ for $p \in \mathcal{P}$ if $\mathcal{P}$ is not empty and 0 otherwise. Furthermore, a constraint is said to be "free" if, for any $p \in \mathcal{P}$, the only edges in $\boldsymbol{\gamma}(p)$ with weights different from $-\infty$ are edges of the forms (i) $\sigma_p^t \xrightarrow{k_{(\sigma_p^t, \omega_p^u)}} \omega_p^u$, (ii) $\sigma_p^t \xrightarrow{k_{(\sigma_p^t)}} 0$, or (iii) $\omega_p^t \xrightarrow{k_{(\omega_p^t)}} 0$ for some $t, u \in \texttt{Reg}(p, \boldsymbol{\gamma}(p))$ and $k_{(\sigma_p^t, \omega_p^u)}, k_{(\sigma_p^t)}, k_{(\omega_p^t)} \in \mathbb{N}$*

Free constraints are only allowed to impose, for the same phaser, non-negative lower bounds on differences between signals and waits, between signals and 0, and between waits and 0. Like degree-0-constraints, free constraints are not allowed to put a positive upper bound on how much a signal is larger than a wait. Unlike degree-0-constraints, they are not allowed to put bounds on the differences among signal values, or among wait values. For instance a free constraint cannot impose $\sigma_p^t - \sigma_p^u = 0$ while a degree-0-constraint can. Intuitively, freeness does not oblige our verification procedure to maintain exact differences when firing "signal" or "wait" instructions, jeopardizing termination. This will be stated in Section VI.

C. Denotations of constraints.

Given a configuration $c = (\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}, \boldsymbol{pv}, \boldsymbol{\varphi})$ and a constraint $\phi = (\mathcal{T}', \mathcal{P}', \boldsymbol{bv}', \boldsymbol{pc}', \boldsymbol{pv}', \boldsymbol{\gamma}')$, we say that c satisfies ϕ, and write $c \models \phi$, if c satisfies (up to a renaming of the tasks and the phasers) conditions imposed by ϕ. More concretely, $c \models \phi$ if $\boldsymbol{bv} = \boldsymbol{bv}'$ and there are bijections $\tau : \mathcal{T} \to \mathcal{T}'$ and $\pi : \mathcal{P} \to \mathcal{P}'$ such that: (i) $\boldsymbol{pc}(t) = \boldsymbol{pc}'(\tau(t))$ for each $t \in \mathcal{T}$; and (ii) $\pi(\boldsymbol{pv}(t)(\mathtt{v})) = \boldsymbol{pv}'(\tau(t))(\mathtt{v})$ for each $t \in \mathcal{T}$ and $\mathtt{v} \in \mathtt{V}$; and (iii) the renaming of tasks and phasers in $\boldsymbol{\varphi}$ wrt. τ and π satisfies $\boldsymbol{\gamma}$, i.e., (iii.a) for each $t \in \mathcal{T}$ and each $p \in \mathcal{P}$, $\boldsymbol{\varphi}(p)(t) \downarrow$ iff $\texttt{isReg}(\tau(t), \pi(p), \boldsymbol{\gamma}(\pi(p)))$, and (iii.b) for each $p' \in \mathcal{P}'$, $\wp(\bigwedge_{t' \in \texttt{Reg}(p', \boldsymbol{\gamma}(p'))}((\omega_{p'}^{t'}, \sigma_{p'}^{t'}) = \boldsymbol{\varphi}(\pi^{-1}(p'))(\tau^{-1}(t')))) \models \boldsymbol{\gamma}(p')$. We let $[\![\phi]\!]$ denote $\{c \mid c \models \phi\}$. Intuitively, $[\![(\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}, \boldsymbol{pv}, \boldsymbol{\gamma})]\!]$ contains all configurations c with the same number of tasks and phasers and such that there are renamings of tasks and phasers that preserve in c the correspondence between $\boldsymbol{pc}$, $\boldsymbol{pv}$ and $\boldsymbol{\gamma}$. We write $[\![\Phi]\!]$, for a set Φ of constraints, to mean the union $\cup_{\phi \in \Phi} [\![\phi]\!]$. Given a program $(\mathtt{B}, \mathtt{V}, \mathtt{T})$, we can exactly characterize with a finite set of constraints all configurations involving n tasks and p phasers and satisfying the premises of rules (`runtime error`), (`assert. fault`), (`race`) and (`deadlock`) from Fig.(3).

Lemma 1 (Characterizing badness)**.** *Given a program $(\mathtt{B}, \mathtt{V}, \mathtt{T})$ and natural numbers (n, p), we can exhibit finite sets of constraints $\texttt{badCstrs}_{\texttt{race}}^{(n,p)}$, $\texttt{badCstrs}_{\texttt{assert}}^{(n,p)}$, $\texttt{badCstrs}_{\texttt{runtime}}^{(n,p)}$ and $\texttt{badCstrs}_{\texttt{deadlock}}^{(n,p)}$ such that:*

$$\texttt{badConfs}_{\texttt{race}}^{(n,p)} = [\![\texttt{badCstrs}_{\texttt{race}}^{(n,p)}]\!]$$
$$\texttt{badConfs}_{\texttt{assert}}^{(n,p)} = [\![\texttt{badCstrs}_{\texttt{assert}}^{(n,p)}]\!]$$
$$\texttt{badConfs}_{\texttt{runtime}}^{(n,p)} = [\![\texttt{badCstrs}_{\texttt{runtime}}^{(n,p)}]\!]$$
$$\texttt{badConfs}_{\texttt{deadlock}}^{(n,p)} = [\![\texttt{badCstrs}_{\texttt{deadlock}}^{(n,p)}]\!]$$

In addition, we can choose the constraints in $\texttt{badCstrs}_{\texttt{deadlock}}^{(n,p)}$ to be of degree 0 while those in $\texttt{badCstrs}_{\texttt{race}}^{(n,p)}$, $\texttt{badCstrs}_{\texttt{assert}}^{(n,p)}$ or in $\texttt{badCstrs}_{\texttt{runtime}}^{(n,p)}$ to be free.

D. Entailment.

We say that a constraint $\phi = (\mathcal{T}, \mathcal{P}, \boldsymbol{bv}, \boldsymbol{pc}, \boldsymbol{pv}, \boldsymbol{\gamma})$ is weaker than a constraint $\phi' = (\mathcal{T}', \mathcal{P}', \boldsymbol{bv}', \boldsymbol{pc}', \boldsymbol{pv}', \boldsymbol{\gamma}')$, written $\phi \sqsubseteq \phi'$, to mean the following. First, the two constraints have the same number of phasers and tasks, agree on the values of the Boolean variables and, up to renamings, on the values of the phaser variables and on which tasks are registered to which phasers. Second, the constraints on the wait and signal values are stronger in ϕ' than in ϕ. More formally, $\phi \sqsubseteq \phi'$ if $\boldsymbol{bv} = \boldsymbol{bv}'$ and there are bijections $\tau : \mathcal{T} \to \mathcal{T}'$ and $\pi : \mathcal{P} \to \mathcal{P}'$ s.t. for each $t \in \mathcal{T}$ and $p \in \mathcal{P}$ the following four conditions hold: (i) $\boldsymbol{pc}(t) = \boldsymbol{pc}'(\tau(t))$; and (ii) $\pi(\boldsymbol{pv}(t)(\mathtt{v})) = \boldsymbol{pv}'(\tau(t))(\mathtt{v})$; and (iii) $\pi(\texttt{Reg}(p, \boldsymbol{\gamma}(p))) = \texttt{Reg}(\pi(p), \boldsymbol{\gamma}'(\pi(p)))$; and (iv) $\boldsymbol{\gamma}(p) \sqsubseteq_{\mathcal{G}} \boldsymbol{\gamma}'(\pi(p))\left[\left\{\omega_{\pi(p)}^{\tau(t)}/\omega_p^t, \sigma_{\pi(p)}^{\tau(t)}/\sigma_p^t \mid t \in \texttt{Reg}(p, \boldsymbol{\gamma}(p))\right\}\right]$. Clearly, $\phi \sqsubseteq \phi'$ implies $[\![\phi']\!] \subseteq [\![\phi]\!]$. We say that $\sqsubseteq$ is sound.

We can show that $\sqsubseteq$ is a well-quasi-order[1] over constraints of bounded degrees and involving fixed numbers of tasks and phasers since $\sqsubseteq_{\mathcal{G}}$ is itself a well-quasi-ordering over graphs of bounded degrees over a finite set of variables ([9], [12]).

Lemma 2 (WQO)**.** *Given $k, n, p \in \mathbb{N}$, the entailment relation $\sqsubseteq$ over the set of constraints of degree k involving at most n tasks and p phasers is a well-quasi-order.*

[1] A reflexive and transitive binary relation $\preceq$ is a well-quasi-order over a set A if there is no infinite sequence $a_0, a_1, \ldots$ of A elements s.t. $a_i \not\preceq a_j$ for all $i < j$.

$$\frac{\boldsymbol{pc}' = \boldsymbol{pc}[t \leftarrow \mathtt{v} := \mathtt{newPhaser}(); \boldsymbol{pc}(t)] \wedge \boldsymbol{pv}(t)(\mathtt{v}) = p \wedge \boldsymbol{pv}' = \boldsymbol{pv}[t \leftarrow \boldsymbol{pv}(t)[\mathtt{v} \leftarrow \uparrow]] \wedge \left\{\omega_p^t \mapsto 0, \sigma_p^t \mapsto 0\right\} \models \boldsymbol{\gamma}(p) \wedge \boldsymbol{\gamma}' = \boldsymbol{\gamma} \setminus \{p\} \wedge (\mathtt{isReg}(u,p,\boldsymbol{\gamma}(p)) \implies u = t)}{(\mathcal{T},\mathcal{P},\boldsymbol{bv},\boldsymbol{pc},\boldsymbol{pv},\boldsymbol{\gamma}) \overset{t}{\rightsquigarrow} (\mathcal{T},\mathcal{P}\setminus\{p\},\boldsymbol{bv},\boldsymbol{pc}',\boldsymbol{pv}',\boldsymbol{\gamma}')} \text{(newPhaser I)}$$

$$\frac{\boldsymbol{pc}' = \boldsymbol{pc}[t \leftarrow \mathtt{v} := \mathtt{newPhaser}(); \boldsymbol{pc}(t)] \wedge \boldsymbol{pv}(t)(\mathtt{v}) = p \wedge \boldsymbol{pv}' = \boldsymbol{pv}[t \leftarrow \boldsymbol{pv}(t)[\mathtt{v} \leftarrow q]] \wedge \left\{\omega_p^t \mapsto 0, \sigma_p^t \mapsto 0\right\} \models \boldsymbol{\gamma}(p) \wedge \boldsymbol{\gamma}' = \boldsymbol{\gamma} \setminus \{p\} \wedge (\mathtt{isReg}(u,p,\boldsymbol{\gamma}(p)) \implies u = t)}{(\mathcal{T},\mathcal{P},\boldsymbol{bv},\boldsymbol{pc},\boldsymbol{pv},\boldsymbol{\gamma}) \overset{t}{\rightsquigarrow} (\mathcal{T},\mathcal{P}\setminus\{p\},\boldsymbol{bv},\boldsymbol{pc}',\boldsymbol{pv}',\boldsymbol{\gamma}')} \text{(newPhaser II)}$$

$$\frac{\boldsymbol{pc}' = \boldsymbol{pc}[t \leftarrow \mathtt{v.signal}(); \boldsymbol{pc}(t)] \wedge \boldsymbol{pv}(t)(\mathtt{v}) = p \wedge \mathtt{isReg}(t,p,\boldsymbol{\gamma}(p)) \wedge \wp = \left(\boldsymbol{\gamma}(p) \otimes \mathtt{graphOf}\left(\wedge_{u \in \mathtt{Reg}(p,\boldsymbol{\gamma}(p))}(\sigma_p^t > \omega_p^u \geq 0)\right)\right) \wedge \mathtt{isSat}(\wp) \wedge \boldsymbol{\gamma}' = \boldsymbol{\gamma}\left[p \leftarrow \left(\left(\wp\left[\sigma_p^t/\sigma\right] \otimes \mathtt{graphOf}\left(\sigma_p^t = \sigma - 1\right)\right) \ominus \{\sigma\}\right)\right]}{(\mathcal{T},\mathcal{P},\boldsymbol{bv},\boldsymbol{pc},\boldsymbol{pv},\boldsymbol{\gamma}) \overset{t}{\rightsquigarrow} (\mathcal{T},\mathcal{P},\boldsymbol{bv},\boldsymbol{pc}',\boldsymbol{pv},\boldsymbol{\gamma}')} \text{(signal)}$$

$$\frac{\boldsymbol{pc}' = \boldsymbol{pc}[t \leftarrow \mathtt{v.wait}(); \boldsymbol{pc}(t)] \wedge \boldsymbol{pv}(t)(\mathtt{v}) = p \wedge \mathtt{isReg}(t,p,\boldsymbol{\gamma}(p)) \wedge \wp = \left(\boldsymbol{\gamma}(p) \otimes \mathtt{graphOf}\left(\wedge_{\{u \in \mathtt{Reg}(p,\boldsymbol{\gamma}(p))\}}(\sigma_p^u \geq \omega_p^t > 0)\right)\right) \wedge \mathtt{isSat}(\wp) \wedge \boldsymbol{\gamma}' = \boldsymbol{\gamma}\left[p \leftarrow \left(\left(\wp\left[\omega_p^t/\omega\right] \otimes \mathtt{graphOf}\left(\omega_p^t = \omega - 1\right)\right) \ominus \{\omega\}\right)\right]}{(\mathcal{T},\mathcal{P},\boldsymbol{bv},\boldsymbol{pc},\boldsymbol{pv},\boldsymbol{\gamma}) \overset{t}{\rightsquigarrow} (\mathcal{T},\mathcal{P},\boldsymbol{bv},\boldsymbol{pc}',\boldsymbol{pv},\boldsymbol{\gamma}')} \text{(wait)}$$

$$\frac{\boldsymbol{pc}' = \boldsymbol{pc}[t \leftarrow \mathtt{v.drop}(); \boldsymbol{pc}(t)] \wedge \boldsymbol{pv}(t)(\mathtt{v}) = p \wedge \neg\mathtt{isReg}(t,p,\boldsymbol{\gamma}(p)) \wedge \boldsymbol{\gamma}' = \boldsymbol{\gamma}\left[p \leftarrow \left(\boldsymbol{\gamma}(p) \otimes \mathtt{graphOf}\left((\sigma_p^t \geq \omega_p^t \geq 0) \wedge_{u \in \mathtt{Reg}(p,\boldsymbol{\gamma}(p))} (\sigma_p^u \geq \omega_p^t \geq 0) \wedge (\sigma_p^t \geq \omega_p^u \geq 0)\right)\right)\right]}{(\mathcal{T},\mathcal{P},\boldsymbol{bv},\boldsymbol{pc},\boldsymbol{pv},\boldsymbol{\gamma}) \overset{t}{\rightsquigarrow} (\mathcal{T},\mathcal{P},\boldsymbol{bv},\boldsymbol{pc}',\boldsymbol{pv},\boldsymbol{\gamma}')} \text{(drop)}$$

$$\frac{\begin{array}{c}\boldsymbol{pc}' = \boldsymbol{pc}[t \leftarrow \mathtt{asynch}(\mathtt{task},\mathtt{v}_1,\ldots\mathtt{v}_k)\{\mathtt{s}_1\}; \boldsymbol{pc}(t)] \wedge \mathtt{paramOf}(\mathtt{task}) = (\mathtt{w}_1,\ldots\mathtt{w}_k) \wedge u \in \mathcal{T}\setminus\{t\} \wedge \\ \boldsymbol{pv}' = \boldsymbol{pv}\setminus\{u\} \wedge \boldsymbol{pc}(u) = \mathtt{s}_1 \wedge \forall i : 1 \leq i \leq k.\ \boldsymbol{pv}(t)(\mathtt{v}_i) = \boldsymbol{pv}(u)(\mathtt{w}_i) = p_i \wedge (\mathtt{isReg}(t,p_i,\boldsymbol{\gamma}(p_i)) \Leftrightarrow \mathtt{isReg}(u,p_i,\boldsymbol{\gamma}(p_i))) \wedge \\ \wp_i = \left(\boldsymbol{\gamma}(p_i) \otimes \mathtt{graphOf}\left(\omega_{p_i}^t = \omega_{p_i}^u \wedge \sigma_{p_i}^t = \sigma_{p_i}^u\right)\right) \wedge \mathtt{isSat}(\wp_i) \wedge \boldsymbol{\gamma}_0 = \boldsymbol{\gamma} \wedge \boldsymbol{\gamma}_i = \boldsymbol{\gamma}_{i-1}\left[p_i \leftarrow \left(\wp_i \ominus \left\{\omega_{p_i}^u, \sigma_{p_i}^u\right\}\right)\right]\end{array}}{(\mathcal{T},\mathcal{P},\boldsymbol{bv},\boldsymbol{pc},\boldsymbol{pv},\boldsymbol{\gamma}) \overset{t}{\rightsquigarrow} (\mathcal{T}\setminus\{u\},\mathcal{P},\boldsymbol{bv},\boldsymbol{pc}'\setminus\{u\},\boldsymbol{pv}',\boldsymbol{\gamma}_n)} \text{(asynch)}$$

$$\frac{t \notin \mathcal{T} \wedge \boldsymbol{pc}' = \boldsymbol{pc}[t \leftarrow \mathtt{exit}] \wedge f \in \mathtt{Pfn}(\mathtt{V},\mathcal{P}) \wedge \boldsymbol{pv}' = \boldsymbol{pv} \uplus \{t \mapsto f\} \wedge \mathcal{Q} \subseteq \mathcal{P} \wedge \boldsymbol{\gamma}' = \boldsymbol{\gamma}\left[\left\{p \leftarrow \boldsymbol{\varphi}(p) \otimes \mathtt{graphOf}\left((\sigma_p^t \geq \omega_p^t \geq 0) \wedge_{u \in \mathtt{Reg}(p,\boldsymbol{\gamma}(p))} (\sigma_p^u \geq \omega_p^t \geq 0) \wedge (\sigma_p^t \geq \omega_p^u \geq 0)\right) \mid p \in \mathcal{Q}\right\}\right]}{(\mathcal{T},\mathcal{P},\boldsymbol{bv},\boldsymbol{pc},\boldsymbol{pv},\boldsymbol{\gamma}) \overset{t}{\rightsquigarrow} (\mathcal{T}\cup\{t\},\mathcal{P},\boldsymbol{bv},\boldsymbol{pc}',\boldsymbol{pv}',\boldsymbol{\gamma}')} \text{(exit)}$$

Fig. 4. Derivation rules for computing $\mathtt{pre}(t,\phi)$ for phaser statements as union of all $\{\phi' \mid \phi \overset{t}{\rightsquigarrow} \phi'$ with $\phi = (\mathcal{T},\mathcal{P},\boldsymbol{bv},\boldsymbol{pc},\boldsymbol{pv},\boldsymbol{\gamma})$ and $t \in \mathcal{T}\}$. Derivations for other program statements are straightforward.

VI. Verification Procedure

Input: A program $\mathtt{prg} = (\mathtt{B},\mathtt{V},\mathtt{T})$, a set Φ_{bad} of pairwise $\sqsubseteq$-incomparable constraints, maximum upper bounds $t^\bullet$ and $p^\bullet$ (in $\mathbb{N}\cup\{+\infty\}$) on coexisting tasks and phasers.
Output: A symbolic run to Φ_{bad} or the value `unreachable`

1 Initialize both `Working` and `Visited` to $\{(\phi,\phi) \mid \phi \in \Phi_{bad}\}$;
2 **while** *there exists* $(\phi,\tau) \in$ `Working` **do**
3 remove (ϕ,τ) from `Working`;
4 let $(\mathcal{T},\mathcal{P},\boldsymbol{bv},\boldsymbol{pc},\boldsymbol{pv},\boldsymbol{\gamma}) = \phi$;
5 **if** $|\mathcal{T}| > t^\bullet$ *or* $|\mathcal{P}| > p^\bullet$ **then** `continue`;
6 **if** $c_{init} \models \phi$ **then return** τ;
7 **foreach** $t \in \mathcal{T}$ **do**
8 **foreach** $\phi' \in \mathtt{pre}(t,\phi)$ **do**
9 **if** $\psi \not\sqsubseteq \phi'$ *for all* $(\psi,\_) \in$ `Visited` **then**
10 Remove from `Working` and `Visited` each $(\psi,\_)$ for which $\phi' \sqsubseteq \psi$;
11 Add $(\phi',\phi' \cdot t \cdot \tau)$ to both `Working` and `Visited`;
12 **return** *unreachable* ;

Procedure check($\mathtt{prg}$,Φ_{bad},$t^\bullet$, $p^\bullet$), a simple working list procedure for checking constraints reachability.

We discuss in the following the procedure *check* depicted above and assume a program prg and a set Φ_{bad} of constraints the reachability of which we want to check. Φ_{bad} can for example be any subset of $\mathtt{badCstrs}^{(n,p)}_{\mathtt{deadlock}}$ (degree 0) or of $\mathtt{badCstrs}^{(n,p)}_{\mathtt{assert}}$ (free) in case we want to check the possibility of a deadlock or of an assertion violation.

It is not difficult to show that $[\![\mathtt{pre}(t,\phi)]\!]$ (obtained as described in Fig.(4)) coincides with $\{c' \mid c' \overset{t}{\rightarrow} c$ and $c \in [\![\phi]\!]\}$. Using the soundness of $\sqsubseteq$, we can show by induction the partial correctness of the procedure check($\mathtt{prg}$,Φ_{bad},$+\infty$,$+\infty$).

Lemma 3 (Partial correctness)**.** *If* check($\mathtt{prg}$,Φ_{bad},$+\infty$,$+\infty$) *returns* `unreachable`*, then* $c_{init} \not\overset{*}{\rightarrow} [\![\Phi_{bad}]\!]$*. If it returns a trace* $\phi_n \cdot t_n \cdots t_1 \cdot \phi_1$ *then there are* $c_n,\ldots c_1$ *with* $c_n = c_{init}$*,* $c_1 \in [\![\Phi_{bad}]\!]$ *and* $c_i \overset{t_i}{\rightarrow} c_{i-1}$ *for* $i : 1 < i \leq n$*.*

Theorem 2 (Free termination)**.** check($\mathtt{prg}$,Φ_{bad},$t^\bullet$,$p^\bullet$) *terminates for* $t^\bullet, p^\bullet \in \mathbb{N}$ *and free* Φ_{bad}*.*

Proof. Sketch. Freeness is preserved by the pre computation (Fig.(4)). Suppose the procedure does not terminate. The infinite sequence of constraints passing the test at line 9 of the procedure violates well-quasi-orderness of $\sqsubseteq$ over free constraints with fixed numbers of tasks and phasers. □

In order to check reachability of arbitrary constraints, we may need to force termination. We do this by soundly bounding the degree of generated constraints using a relaxation ρ_k. The relaxation $\rho_k((\mathcal{T},\mathcal{P},\boldsymbol{bv},\boldsymbol{pc},\boldsymbol{pv},\boldsymbol{\gamma}))$ replaces, in each graph $\boldsymbol{\gamma}(p)$, each weight k'' s.t. $k'' < -k$ with $-\infty$.

1 **foreach** $\phi'' \in \mathtt{pre}(t,\phi)$ **do**
2 Let $\phi' = \rho_k(\phi'')$;

Fig. 5. Systematic relaxation

Theorem 3 (Forced termination)**.** *Procedure* check(prg,Φ_{bad},$t^\bullet$,$p^\bullet$) *for* $t^\bullet, p^\bullet \in \mathbb{N}$*, with line 8 replaced by the lines of Fig. (5), is sound and guaranteed to terminate.*

Proof. Soundness is due to the validity of $\rho_k(\phi) \sqsubseteq \phi$ while the termination argument relies, similarly to Theorem (2), on well-quasi orederness of $\sqsubseteq$ on the set of constraints with bounded degree and fixed numbers of tasks and phasers. □

VII. Experimental Results

We report on experiments with our open source prototype *hjVerify* (https://gitlab.ida.liu.se/apv/hjVerify) for the verification of phaser programs. We conducted experiments on 12 different programs (some of which are from [5]). We considered both deadlock and assertion reachability problems. For each property, we considered correct and buggy versions. This gave 48 different instances with 2 to 3 phasers and 2 to 4 tasks (except for the parameterized case). Our tool uses global phaser and task variables as in [5]. We have experimented with adapting the view abstraction technique [14] to verify phaser programs generating arbitrary many tasks, i.e., parameterized verification where the number of phasers is fixed. (see [11] for more details.) We report on two parameterized examples. Experiments were conducted on a 2.9GHz processor with 8GB of memory.

program	property	safe / buggy	times
01.Loopless	deadlock:	ok / trace	1s / 1s
	assertion:	ok / trace	1s / 1s
02.Iterative averaging	deadlock:	ok / trace	1s / 1s
	assertion:	ok / trace	1s / 1s
03.Ordered phasers	deadlock:	ok / trace	1s / 1s
	assertion:	ok / trace	13s / 1s
04.Conditional	deadlock:	ok / trace	2s / 1s
	assertion:	ok / trace	4s / 7s
05.Loop Synch.	deadlock:	ok / trace	178s / 145s
	assertion:	ok / trace	7s / 13s
06.Nested forks	deadlock:	ok / trace	2s / 1s
	assertion:	ok / trace	1s / 1s
07.Conditional membership	deadlock:	ok / trace	1s / 1s
	assertion:	ok / trace	12s / 3s
08.Producer-consumer	deadlock:	ok / trace	37s / 222s
	assertion:	ok / trace	79s / 34s
09.Parameterized loopless	deadlock:	ok / trace	20s / 1s
	assertion:	ok / trace	67s / 1s
10.Parameterized iterative-averaging	deadlock:	ok / trace	1s / 1s
	assertion:	ok / trace	1s / 1s
11.Running-2	deadlock:	ok / trace	5s / 1s
	assertion:	ok / trace	26s / 4s
12.Running-3	deadlock:	ok / trace	4318s / 128s
	assertion:	ok / trace	18631s / 54s

Our implemented procedure does not eagerly concretize all task states as described in the predecessor computation of Section V. Instead we collect conditions on the phases of the tasks that did not take any action yet and lazily concretize them. Reported times for checking deadlocks are the sums of the times required to check reachability for each cycle. The prototype is only a proof of concept. For instance, the example (12.Running-3) is a variant of (11-Running-2) where a task instance is spawned twice leading to two symmetrical tasks (out of four). This required up to three orders of magnitude more time to check. We believe partial order reduction techniques would help here. Other relevant heuristics would be to make use of priority queues and to organize the minimal sets. All examples are available on the tool homepage.

VIII. Conclusion

We have proposed a gap-order based reachability analysis for phaser programs. We have showed our analysis to be exact and guaranteed to terminate when checking runtime, race and assertion errors. We have established the undecidability of deadlock verification and explained how to turn our analysis into a sound over-approximation. To the best of our knowledge, this is beyond the capabilities of current verification techniques which currently only target concrete inputs to phaser programs. We are currently working on tackling the parameterized case and have obtained preliminary encouraging results. Apart from improving the scalability of the tool and from using it in combination with predicate abstraction and abstract interpretation in order to analyze actual source code, we are investigating the applicability of the presented techniques for the verification of similar synchronization constructs.

References

[1] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," *SIGPLAN Not.*, vol. 40, no. 10, pp. 519–538, Oct. 2005.

[2] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer, "Phasers: a unified deadlock-free construct for collective and point-to-point synchronization," in *Proceedings of the 22nd annual international conference on Supercomputing*. ACM, 2008, pp. 277–288.

[3] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-java: the new adventures of old x10," in *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*. ACM, 2011, pp. 51–61.

[4] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer, "Phaser accumulators: A new reduction construct for dynamic parallelism," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–12.

[5] T. Cogumbreiro, R. Hu, F. Martins, and N. Yoshida, "Dynamic deadlock verification for general barrier synchronisation," in *20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: ACM, 2015, pp. 150–160.

[6] D.-K. Le, W.-N. Chin, and Y.-M. Teo, "Verification of static and dynamic barrier synchronization using bounded permissions," in *Int. Conf. on Formal Engineering Methods*. Springer, 2013, pp. 231–248.

[7] P. Anderson, B. Chase, and E. Mercer, "Jpf verification of habanero java programs," *SIGSOFT Softw. Eng. Notes*, vol. 39, no. 1, pp. 1–7, 2014.

[8] K. Havelund and T. Pressburger, "Model checking java programs using java pathfinder," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 2, no. 4, pp. 366–381, 2000.

[9] R. Mayr and P. Totzke, "Branching-time model checking gap-order constraint systems," *Fundamenta Informaticae*, vol. 143, no. 3-4, pp. 339–353, 2016.

[10] L. Bozzelli and S. Pinchinat, "Verification of gap-order constraint abstractions of counter systems," *Theoretical Computer Science*, vol. 523, pp. 1 – 36, 2014.

[11] G. Zeinab, R. Ahmed, E. Petru, and P. Zebo, "Safety verification of phaser programs," *CoRR*, vol. abs/1708.02801, 2017. [Online]. Available: https://arxiv.org/abs/1708.02801

[12] P. Z. Revesz, "A closed-form evaluation for datalog queries with integer (gap)-order constraints," *Theoretical Computer Science*, vol. 116, no. 1, pp. 117–149, 1993.

[13] S. Lahiri and M. Musuvathi, "An efficient decision procedure for utvpi constraints," in *Frontiers of Combining Systems (FroCos '05)*. Springer Verlag, May 2005.

[14] P. A. Abdulla, F. Haziza, and L. Holík, "All for the price of few," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2013, pp. 476–495.