Dynamic Deadlock Verification for General Barrier Synchronisation

TIAGO COGUMBREIRO, Rice University, USA
RAYMOND HU, Imperial College London, UK
FRANCISCO MARTINS, LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal and University of the Azores, PT
NOBUKO YOSHIDA, Imperial College London, UK

We present Armus, a verification tool for dynamically detecting or avoiding barrier deadlocks. The core design of Armus is based on phasers, a generalisation of barriers that supports split-phase synchronisation, dynamic membership, and optional-waits. This allows Armus to handle the key barrier synchronisation patterns found in modern languages and libraries. We implement Armus for X10 and Java, giving the first sound and complete barrier deadlock verification tools in these settings.

Armus introduces a novel event-based graph model of barrier concurrency constraints that distinguishes task-event and event-task dependencies. Decoupling these two kinds of dependencies facilitates the verification of distributed barriers with dynamic membership, a challenging feature of X10. Further, our base graph representation can be dynamically switched between a task-to-task model, Wait-for Graph (WFG), and an event-to-event model, State Graph (SG), to improve the scalability of the analysis.

Formally, we show that the verification is sound and complete with respect to the occurrence of deadlock in our core phaser language; and that switching graph representations preserves the soundness and completeness properties. These results are machine checked with the Coq proof assistant. Practically, we evaluate the runtime overhead of our implementations using three benchmark suites in local and distributed scenarios. Regarding deadlock detection, distributed scenarios show negligible overheads and local scenarios show overheads below $1.15\times$. Deadlock avoidance is more demanding, and highlights the potential gains from dynamic graph selection. In one benchmark scenario, the runtime overheads vary from: $1.8\times$ for dynamic selection, $2.6\times$ for SG-static selection, and $5.9\times$ for WFG-static selection.

CCS Concepts:
- Software and its engineering → Deadlocks; Software verification; Dynamic analysis; Concurrent programming structures; Semantics;

Additional Key Words and Phrases: barrier synchronisation, phasers, deadlock detection, deadlock avoidance, X10, Java

ACM Reference Format:

Authors’ addresses: Tiago Cogumbreiro, Rice University, Department of Computer Science, Houston, TX, USA, cogumbreiro@gmail.com; Raymond Hu, Imperial College London, Department of Computing, London, UK, raymond.hu05@imperial.ac.uk; Francisco Martins, LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal and University of the Azores, Faculty of Sciences and Technology, Ponta Delgada, PT, fmartins@acm.org; Nobuko Yoshida, Imperial College London, Department of Computing, London, UK, n.yoshida@imperial.ac.uk.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 Association for Computing Machinery.
0164-0925/2018/1-ART1 $15.00
https://doi.org/10.1145/nmnmmnn.nmmnnmn
1 INTRODUCTION

Dynamic verification of barrier deadlocks. The rise of multicore processors and networked clusters has pushed mainstream programming languages to incorporate various concurrency features, an important class of which are barriers and their related mechanisms. The basic functionality of a barrier is to designate a point in the execution of a group of tasks at which each task is blocked until all have reached the barrier. Java 5–8 and .NET 4, for example, introduced several standard APIs that provide barriers explicitly or are built on top of barriers: latches, cyclic barriers, fork/join, futures, and streams. Recent languages for parallel programming have also been designed with more advanced abstractions as first-class language features, such as clocks in X10 [10] and phasers in Habanero-Java (HJ) [8], that are more expressive than basic barriers.

As with many other concurrency mechanisms, deadlocks—in which two or more tasks blocked on distinct barriers are waiting (perhaps indirectly) for each other—are one of the primary errors arising in barrier programs. Historically, the approach to counter barrier deadlocks has been to restrict the permitted barrier synchronisation patterns such that programs are barrier-deadlock free by construction; e.g., OpenMP\(^1\) restricts barrier composition to syntactic nesting. Unfortunately, to date there are no available tools for comprehensive verification of barrier-deadlocks in X10 or HJ, nor for standard libraries such as the Java Phaser\(^2\) and the .NET Barrier\(^4\) APIs.

Two key issues make barrier-deadlock verification challenging in these recent languages and systems. The first is that barriers may be created dynamically and communicated among tasks as values, referred to as first-class barriers [67]. Due to the difficulty of statically analysing the usage of first-class barriers precisely (e.g., due to aliases and non-determinism), the state-of-the-art in barrier-deadlock verification is based on dynamic techniques that monitor program execution at run-time. (Existing tools for static verification are limited to simpler systems where barriers permit only global, i.e., system-wide, synchronisations; see Section 8.) The second is that, in contrast to the conservative restrictions in earlier systems, the richer barrier features in recent languages are motivated by expressiveness at the cost of making deadlock verification, including dynamic approaches, more complicated. One of the key features supported in Java, .NET, X10, and HJ, but not handled by any existing barrier-deadlock verification tool, is dynamic membership [53], which allows the group of tasks participating in synchronisations on a barrier to change during execution.

The state-of-the-art in dynamic barrier-deadlock verification is based on the well-established concept of Wait-For Graph (WFG) and has been developed for MPI\(^3\) (e.g., the MUST error detection tool [32, 34]) and UPC\(^4\) [57]. A WFG [40] is a graph model of the control flow dependencies between tasks. Applied to barriers, the WFG nodes represent tasks, and a directed edge from a task \(t\) to task \(t'\) signifies that \(t\) is blocked on a barrier, waiting for \(t'\) to reach the barrier. WFG-based approaches typically work by maintaining an abstract representation of the concurrency constraints in a system, from which the WFG can be derived and deadlock detection performed as a suitable graph analysis, such as checking for circular dependencies.

Existing WFG-based tools, such as MUST, offer precise deadlock detection in systems featuring multiple barriers, but suffer from limitations in the presence of more advanced barrier features. One is that they are designed on the assumption of static barrier membership (as is the case for MPI barriers). Naive extension to dynamic membership faces the challenges of maintaining the membership status of barriers consistently (since barrier synchronisations and (de)registration operations occur concurrently) and efficiently (e.g., w.r.t. the overheads of any additional state verification).
synchronisations used for consistency). These issues are exacerbated by distributed barriers, a key design point of X10.

Another limitation is committing exclusively to the WFG model. The WFG originates from a distributed databases setting [40] involving a fixed number of tasks and dynamic resource creation. The WFG was thus optimised for concurrency constraints between fewer tasks and more resources, which is often not the case in more advanced barrier programs with dynamic task spawning and barrier creation, as possible in X10 and Java. Its counterpart, the State Graph (SG) [12] favours scenarios with more tasks than barriers. In general, however, we may expect that the most suitable model cannot be predicted a priori, and that the situation may change as execution proceeds. Committing to a specific graph model may thus hinder the scalability of dynamic deadlock verification.

Armus. This paper presents Armus, a dynamic verification framework for general barrier deadlocks based on phasers. A phaser is a generalisation of the concept of barrier that allows tasks to selectively wait on barrier steps, thus permitting any task to progress to an arbitrary future step (i.e., phase) independently of its peers. (Section 2.3 will give more a detailed overview.) Phasers were originally developed in Habanero-Java (HJ) [8, 62] as an extension of X10 clocks, one of the primary motivations being to support asynchronous producer/consumer patterns. Armus is the first framework to support sound and complete deadlock verification for phasers. The key elements of Armus are as follows.

- We formalise the operations of a core concurrent language with phasers that subsumes the barrier facilities of X10, HJ, and the standard Java/.NET Barrier APIs, including dynamic membership. We characterise phaser deadlocks in our language in terms of dependencies between tasks and synchronisation events on phasers.
- On the basis of the above, we introduce a new model for general barrier-deadlock verification, the Task-Event Graph (TEG). We show that deadlock verification by TEG cycle detection is sound and complete with respect to our characterisation of phaser deadlock.
- We show that a WFG and an SG can be readily derived from a TEG, such that all three models are equivalent w.r.t. the existence of cycles. This promotes a technique to improve the scalability of the graph-based verification, by automatically and dynamically switching between models.
- We implement Armus as Armus-X10 and JArmus. Armus-X10 is the first sound and complete deadlock verification tool for native X10 programs using clocks and finish-barriers. We show how the design of Armus lends well to distributed barriers, as implemented in Armus-X10. JArmus is the corresponding tool for Java programs using the Phaser API (and related barrier APIs) extended with one additional method for explicitly registering tasks with phasers.

Armus proposes the TEG model, with its notion of barrier synchronisation events, firstly as a means to capture the richer concurrency dependencies of phaser systems, in comparison to standard barriers. Modelling explicit synchronisation events arises naturally from the fact that tasks waiting on a given phaser may actually be waiting at arbitrarily different phases. The insight of Armus is to interpret the act of waiting on a particular phase as observing a timestamp, in a similar spirit to Lamport’s logical clocks [41], by which we infer dependencies between a task and not only its current phase event (as with basic barriers), but also all future events.

Secondly, the TEG is an effective model for treating dynamic membership. Generating traditional wait-for dependencies directly between tasks requires synchronised bookkeeping between the blocking status and barrier membership of every such pair of tasks. The commonplace technique for building a WFG is to range over every blocked task and then query the blocked operation for its missing participants. Note, however, that for a barrier with dynamic membership, the complete
set of participants can only be known at the end of synchronisation. Armus avoids this issue by decomposing the dependencies into separate relations between tasks and events—a dependency from a blocked task to an event can be asserted independently of the membership of the relevant phaser. This in turn facilitates the application of Armus to distributed barriers: instead of needing to synchronise the status of two potentially remote tasks for each wait-for dependency, Armus allows the global view of the system to be built from the local status of each blocked task (the event being observed, and the phaser memberships of the task) and the monotonic causal ordering of events.

Dynamically selecting between graph models for deadlock detection, based on the monitored ratio of tasks to barriers, is a novel technique of Armus. The difference on the size of the graph can be dramatic. For instance, in an X10 benchmark PS (see Section 7.3), the average edge count for the WFG model is 789 and the SG is 7, while the average using dynamic model selection is 6 (Table 3); and the average execution times, for deadlock avoidance, are 113s for WFG, 50s for SG, and 34s for dynamic model selection (Figure 9). In all cases of our benchmarks, the automatic model selection performs at least as well (i.e., with negligible overheads) as manually selecting the best fixed model.

Outline. This article revises and extends an earlier version of this work [13] with new material and full proofs of results. Firstly, we include a new section with a comprehensive summary of barrier features found in practice and their deadlock characteristics. Regarding our core phaser language, first proposed in the earlier work, we add primitives for awaiting on a phaser at an arbitrary phase, and awaiting on a phaser by an unregistered task; these are needed to model the full functionality of phasers in HJ. Based on the extended language, we make significant updates to the core definitions of phaser deadlocks and dependency relations both in terminology and technical details. The definitions of Task-Event Graph construction, and Wait-For/State Graph contraction, are also revised technically. Regarding the properties of Armus, we develop new proofs of the soundness and completeness of the verification, and graph model equivalence, according to the updated definitions. We highlight that all definitions and proofs in this paper have been formalised and machine-checked in Coq, which is new to this article. Compared to the earlier work, we offer more detailed explanations and extended discussion of the practical methodology and implementations. The performance evaluation is also updated, including new benchmarks to evaluate deadlock verification using the TEG directly, in addition to the WFG/SG. Throughout the paper, we have extended the discussions and included many new examples with detailed explanations. Lastly, we have updated the related work with recent publications.

The structure of this article is as follows.

Section 2 firstly covers the background to this work. We give barrier programming examples in X10, and summarise a range of barrier programming features found in practice and their deadlock characteristics. Secondly, we explain the concept of phasers, and outline the phaser-based approach of Armus to deadlock verification for general barrier synchronisations.

Section 3 defines the core concurrent language with phasers used by Armus to capture all of the surveyed barrier programming features. We define the notions of global and local phaser deadlocks for Armus systems in terms of dependencies between tasks and synchronisation events.

Section 4 presents the deadlock verification methodology of Armus. We define the derivation of a TEG model from Armus system states, and the transformation of a TEG to the associated WFG or SG.

http://www.cs.columbia.edu/~martha/courses/4130/au13/

```java
// Pre: "a" is an array length I+1, initialised to: 0, 0, 0, ..., I
val c = Clock.make(); // Cyclic barrier
finish { // Join barrier
  for (i in 1..(I-1)) // Spawn I-1 child tasks..
    async clocked(c) {
      for (j in 1..J) { // ..that loop (together) J times
        val l = a(i-1);
        val r = a(i+1);
        c.advance(); // Read step on the clock (cyclic synchronisation)
        a(i) = (l+r)/2;
        c.advance(); // Write step on the clock (cyclic synchronisation)
      }
    }
} // Wait for all child tasks to terminate (join synchronisation)
// Result: "a" holds values: 0, 1, 2, ..., I
```

Section 5 shows the main results of Armus: that a TEG and the associated WFG and SG are equivalent w.r.t. the presence of cycles, and that cycle detection in a TEG derived from a system state is sound and complete w.r.t. the occurrence of deadlocks in the system.

Section 6 presents the implementation of Armus for X10 (Armus-X10) and Java (JArmus). We discuss the application of Armus to distributed barriers, implemented in Armus-X10.

Section 7 performs an extensive performance evaluation of Armus in Java and X10, using the NAS Parallel Benchmark, the Java Grande Forum Benchmark suite, and the HPC Challenge benchmark suite. Overall, the worst-case runtime-factor for deadlock detection is $1.21 \times$, and is often not statistically significant, e.g., in distributed benchmarks.

Section 8 discusses related work and Section 9 concludes.

The Armus project Web page includes the full Coq implementation of the definitions and proofs, full source code for the Armus-X10 and JArmus implementations, and the benchmark scripts and data.

2 BARRIER-BASED PARALLEL PROGRAMMING AND DEADLOCKS

2.1 Cyclic and Join Barriers in X10

We start with an introductory deadlocked parallel program that uses barriers, written in X10. Listing 1 implements a simple parallel iterative averaging algorithm [19, 63] that takes a one-dimensional array of I+1 numbers, for I>1, initialised to 0 except for the last element, which is set to I. The algorithm converges on the sequence of natural numbers from 0 through I by repeatedly updating, in parallel, each of the elements (except first and last) to an average of its neighbours. This example features two kinds of barriers: cyclic barriers, for recurrent synchronisation between a set of ongoing tasks, and join barriers, for synchronisation on the termination of a set of tasks.

A parallel task is spawned by the async statement (Line 5) inside the outer for-loop for each index 1 through I-1. A cyclic barrier, represented by the clock created and assigned to c (an immutable val) on Line 2, is used to coordinate these tasks. Each child task is registered (clocked) to the clock; the parent task is implicitly registered on clock creation. The work performed by each
Table 1. Barrier synchronisation features supported in various languages.

<table>
<thead>
<tr>
<th>Feature</th>
<th>UPC</th>
<th>MPI</th>
<th>Java</th>
<th>.NET</th>
<th>X10</th>
<th>HJ (Armus)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group synchronisation</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Split-phase synchronisation</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Dynamic membership</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Async. producer-consumer</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Task in the inner for-loop is split into read and write steps, delimited by the advance operations on c. A barrier synchronisation is performed by calling advance: the calling task is blocked until every task registered to the clock has called advance. The first advance (Line 9) thus ensures every task i completes the read step of the current j-th iteration, reading the a(i−1) and a(i+1) values, before any can proceed to the write step. The second advance similarly ensures every task has finished the current write step, writing the average of the read values to a(i), before any can proceed to the (j+1)-th iteration.

The finish statement applies a join barrier that blocks the executing task (the parent task) at the end of the finish (Line 14) until all nested tasks (the I−1 child tasks) have terminated.

Deadlock due to advanced barrier features. Certain barrier systems are restricted by design to ensure deadlock-freedom (see Section 2.2). By contrast, incorrect use of more advanced barrier features supported in modern systems such as X10 may give rise to subtle deadlock situations. The above example demonstrates a deadlock related to group synchronisation, where different, but not necessarily disjoint, groups of tasks are registered to separate barriers, and dynamic membership of tasks to barrier groups.

The deadlock arises from every child task being blocked on its first advance call (Line 9) because the parent task never performs the corresponding advance. Instead, the parent task is blocked on the finish, waiting for the child tasks to terminate, establishing a cyclic dependency between the parent and each child task.

For both the clock and finish barriers, each child task is dynamically registered at some execution point after the barrier is created: the clock and finish barriers are created on Lines 2 and 3, and the child tasks are spawned and registered later, one by one, in each iteration of Line 5 by the parent task. Tasks may similarly be dynamically deregistered from a barrier. The natural fix for the above deadlock is to have the parent task perform the deregistration operation on the clock, c.drop(), between Lines 13 and 14. Then the resulting synchronisation groups will be such that all tasks are registered to the finish barrier, but only the child tasks are registered to the clock.

Note: it would be incorrect for the parent task to drop its clock membership prior to spawning all the child tasks (or similarly, if X10 did not implicitly register the parent task on clock creation). This would avoid the deadlock, but also introduce a race condition between the collective iteration of the child tasks due to the concurrency between the running tasks and any remaining spawns by the parent.

2.2 An Overview of Barrier Synchronisation Features and Deadlock Errors

We give an overview of a range of key barrier synchronisation features, as supported by the languages and libraries in Table 1. We briefly discuss the purpose of each feature, and the implications for deadlock detection, with small examples. Table 2 summarises up front the support for deadlock verification currently available to programmers in each setting. Prior to Armus, there were no
Table 2. Available support for (dynamic) deadlock verification with respect to the barrier features in Table 1. (‘✓’ means both sound and complete; ‘−’ means no support.)

<table>
<thead>
<tr>
<th></th>
<th>UPC</th>
<th>MPI (MUST)</th>
<th>Java</th>
<th>.NET</th>
<th>X10</th>
<th>HJ</th>
<th>Armus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barrier deadlock</td>
<td>✓</td>
<td>✓ (for group sync. only)</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>sound (but incomplete)</td>
<td>✓</td>
</tr>
<tr>
<td>verification?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

comprehensive barrier deadlock detection facilities or tools, dynamic or otherwise, that supports all of the listed features.

The languages/libraries in Tables 1 and 2 are as follows. **UPC** refers to the barrier functionality of Berkeley Unified Parallel C. **MPI** refers to the MPI_Barrier and MPI_IBarrier operations on an MPI communicator. **X10** refers to clocks (whose functionality subsumes that of finish-barriers and SPMDBarrier). **HJ** (and **Armus**) refer to the functionality of phasers, explained later in Section 2.3. **Java** refers to the standard Phaser API, which is a limited version of the general concept of phasers (see Section 6.4), but nevertheless subsumes the capabilities of other standard Java barrier libraries such as CyclicBarrier and CountDownLatch. **.NET** refers to the standard Barrier API, which is the .NET counterpart of the Java Phaser.

**Group synchronisation.** Barrier programming in UPC is restricted to conducting “global” synchronisations, where every task in a system is implicitly a member of every barrier. A system that only involves such global barriers is deadlock-free if and only if all tasks synchronise on the same barriers in the same order: simply detecting that any two tasks are blocked on different barriers is sufficient to conclude a global deadlock, i.e., that every task in the whole system is (or will become) stuck [57].

Group synchronisation, supported in every other case of Table 1, is the generalisation that permits an arbitrary subset of system tasks to be registered to a barrier. This introduces the notion of local deadlocks, where a subset of system tasks can never progress despite the progress of the system as a whole. The finer granularity of this feature is more expressive, but deadlock detection in turn requires checking for a more general form of circular control flow dependencies between multiple tasks transitively.

The standard approach in practice is to model the concurrency constraints of a system as a Wait-For Graph (WFG), in which circular dependencies manifest as graph cycles. When applied to barriers, the nodes of a WFG represent the tasks, and the edges represent the task-to-task wait-for relation induced by a task being blocked on a barrier that a co-member task has not yet reached. For MPI, the MUST tool includes sound and complete deadlock detection for barriers with group synchronisation (since every MPI communicator is an implicit barrier group) by a WFG approach [32, 34].

**Example 2.1.** In Figure 1 (a), three tasks each synchronise on a different subset of two clocks out of three (a, b, and c). Once all three tasks are spawned, each barrier is synchronising on a different group of two tasks. The order in which each task advances its clocks establishes a circular dependency between all three tasks involving all three clocks, i.e., \( t_1 \) wait-for \( t_2 \) via a, \( t_2 \) wait-for \( t_3 \) via b, and \( t_3 \) wait-for \( t_1 \) via c.
async clocked(a, c) { // t1
    a.advance(); // ×
    c.advance();
}

async clocked(b, a) { // t2
    b.advance(); // ?
    a.advance();
}

async clocked(c, b) { // t3
    c.advance(); // ×
    b.advance();
    a.advance(); // ?
}

if (condition) {
    b.resume();
}

b.advance(); // ?

if (condition) {
    b.drop();
}

a.advance(); // ?

b.advance(); // ?

a.advance(); // ?

a.advance(); // ?

b.advance(); // ?

b.advance(); // ?

/* ? means potential deadlock, depending on the if-condition*/

(a) group synchronisation
(b) split-phase sync.
(c) dynamic membership

Fig. 1. Barrier deadlock Examples 2.1 to 2.3 in X10.

The above example by itself is a global deadlock (the main X10 task, that spawns the three child tasks, implicitly waits for the termination of all spawned tasks). This fragment could, however, constitute a local deadlock as part of a larger system (e.g., by extending the parent task with some continuation), which we formalise in Section 3.2. Dynamic verification of global deadlock is trivial for any system, by simply checking if all (user) tasks are blocked. Many systems implement only global deadlock detection, such as HJ [36], giving a verification that is sound (no false positives) but incomplete (since any non-global deadlock is a false negative).

Split-phase synchronisation. Split-phase synchronisation [29, 38] allows a task to perform a barrier synchronisation over two steps, instead of just a single atomic action. A task initiates its next synchronisation via a non-blocking background operation, and can wait for the synchronisation to conclude, i.e., when all tasks have initiated the synchronisation, as a separate operation at a later point. In X10, the initiation operation on clocks is resume, and the wait operation is simply advance (an advance basically includes an implicit resume if not already performed). Split-phase allows a task to concurrently overlap barrier synchronisation with other work, which is useful for, e.g., hiding network latency in distributed programs [9, 11, 74].

Example 2.2. In Figure 1 (b), the parent and child tasks synchronise on clocks a and b. Although the two tasks wait on their two clocks in opposite order, there is no deadlock if the condition in the child task evaluates to true as it will initiate the synchronisation on b before waiting on a. Otherwise, a deadlock will occur.

Although split-phase synchronisations are available in every case of Table 1, deadlock detection for split-phase is only supported in UPC [57], facilitated by the restriction to global synchronisations described earlier. In the presence of group synchronisation, a verification would have to consider, in addition to blocking status of tasks and group memberships, the initiation status of each task for every synchronisation operation it is involved in. MPI supports split-phase by the immediate MPI_IBARRIER operation, but the deadlock detection in MUST does not take the initiation status of such synchronisations into account [30].

Dynamic membership. A barrier restricted to static membership does not permit the registration or deregistration of tasks once any member task has commenced execution; in the presence of group
synchronisation, static task groups are typically fixed on barrier creation (e.g., MPI communicators). Conversely, dynamic membership allows tasks to be registered and deregistered over the lifetime of a barrier. All of the X10 examples seen so far implicitly feature dynamic membership.

Example 2.3. In Figure 1 (c), the child task is dynamically registered to pre-existing clocks a and b, which the parent task is also registered to (in X10, a task may only register a child task to a clock if it itself is registered). If the condition in the parent task evaluates to false, a deadlock arises because the parent task blocks by advance on a, while the child task blocks on b, establishing a circular dependency. Otherwise, the parent dynamically deregisters from b, allowing the child task to successfully terminate, regardless of whether the deregistration occurs before or after the child reaches the advance. On termination, the child implicitly deregisters from its barriers, which in turn allows the parent task to pass its advance.

Static membership simplifies dynamic analysis of barrier deadlocks because, for any task waiting on some barrier, the set of candidate tasks which the former task may be waiting for (i.e., its co-members) is a run-time invariant throughout the life-cycle of the barrier. This facilitates WFG-based approaches because the only information required to establish a wait-for dependency from a waiting task to any of its fixed co-member tasks is that the latter has not reached the same barrier. By contrast, in dynamic membership the set of wait-for candidates can change per phase. A wait-for dependency to some other task t first requires confirming that the t is indeed a member of the relevant barrier at the point in system execution for which the analysis is being conducted. The difficulty of such checks is compounded when verifying distributed programs, as the query might involve communication between sites to transmit the membership status.

We discuss how Armus supports distributed barriers in Section 6.3. Deadlock detection for distributed barriers is supported by MUST for the static membership barriers of MPI (and non-split-phase synchronisations) [31, 33]; and by UPC, which is restricted to global synchronisations. X10 supports distributed barriers as a key language feature, but without any facility for deadlock detection.

2.3 Generalised Barrier Synchronisation using Phasers

Phasers [8, 62] are a generalisation of barriers. The main feature of phasers is that member tasks may independently progress ahead to a future barrier step (i.e., phase) without synchronising (i.e., waiting for their co-members) on the intermediate steps. Phasers may also support certain usages by non-member tasks.

In this work, we introduce a core language for phaser-coordinated concurrent systems that distills the key functionality of phasers:

- A phaser records the phase, an integer \( n \geq 0 \), reached by each member task.
- A non-blocking arrv operation on a phaser increments the phase of the calling task. Any member task may thus independently advance up to an arbitrary phase.
- A separate, optional await operation on a phaser blocks the calling task at its current phase until every member task has reached this phase.
- Explicit dynamic membership: tasks are registered and deregistered with a phaser by, and only by, reg and dereg operations.

We develop the deadlock verification of Armus on this basis of this core phaser functionality, which subsumes all of the barrier features summarised in Section 2.2 (Table 1). Armus is thus applicable to all of the languages and libraries discussed there, and to any other barrier system whose functionality can be encoded into these operations.
Listing 2. An asynchronous linear producer-consumer pipeline in a pseudo X10 extended with Armus phaser operations (Example 2.4).

```java
// Pre: "ph" is an array of length #num_tasks of new phasers
for (i in 0...(num_tasks-1)) {
    val id = i;
    val local = ph(id);
    val pred = ph((id-1+num_tasks)%num_tasks); // Preceding pipeline task's phaser
    async phased(local, pred) { // Spawned task is registered to these two phasers
        for (j = 1; j <= num_steps; j++) {
            if (id > 0) {
                arrv(pred); // Await next item from preceding task
            } else {
                step(id, j); // Consume preceding task's next item and/or...
                // ...produce next local item as appropriate
            }
            if (id < num_tasks-1) {
                arrv(local); // Signal item is ready
            }
        }
    }
    // Result: every task except #0 has awaited #num_steps on preceding task's phaser,
    // every task except #num_tasks-1 has arrived #num_steps on its local phaser
}
```

**Producer-Consumer.** A primary motivation for phasers is to support asynchronous producer-consumer patterns [14, 64] that cannot be expressed using the barrier features discussed in Section 2.2. Such synchronisation patterns occur in programs performing streaming (also known as data-flow) communication among tasks [54, 60, 63]. A typical producer-consumer application using phasers correlates each phase to the production of one item: a producer task arrives at the next phase after producing an item, while consumer tasks arrive and await the phases in sequence. The ability to advance its local phase without awaiting allows the producer to proceed ahead of the consumers, and similarly allows different consumers to progress at different rates. By contrast, a clocked production stream using, e.g., basic cyclic barriers, would require the producer and every consumer to synchronise on every item.

**Example 2.4.** This example is extracted from the LU (Lower-Upper symmetric Gauss-Seidel) benchmark of the NAS Parallel Benchmark (NPB) suite [24] (see Section 7), originally written in Java using condition variables.10 Listing 2 adapts the example to phasers, which we write in an X10-based pseudo code (for readability) extended with the Armus `arrv` and `await` operations.

The child tasks spawned by the `async` are organised to form a linear pipeline of tasks consuming from one neighbour and producing to the other. Each task is associated with a phaser, stored in an array `ph`, that counts the number of items it has produced locally. Each task, except the first (0-th), awaits the next phase of the preceding task's phaser (to consume the next available item). Each task, except the last, then asynchronously advances its local phaser (after producing the next item) without waiting. (The details of item production and consumption and any additional work performed by the `step` method are omitted for brevity.)

---

10The limited `java.util.concurrent.Phaser` implementation of phasers supports split-phase synchronisations, but not the fully unrestricted phase advancing required for asynchronous producer-consumer.
Incorrect manipulation of task indexes (a typical programming error) in the conditional expressions before and after step can easily give rise to deadlock; e.g., if any task (except the last) does not advance its phaser, or if the first task cyclically waits on the last task’s phaser.

The functionality of phasers modelled in Armus subsumes the previously discussed barrier features as follows. Dynamic membership is supported by the explicit per-task registration and deregistration operations, which in turn supports group synchronisation by allowing an arbitrary subset of tasks to be registered to any given phaser. Split-phase synchronisation is subsumed as the special case of phaser usage where, for each member task, an arrv on phaser p is always followed by an await on p before any subsequent arrv on p; i.e., the discrepancy between the phases of the most and least advanced member tasks of a phaser is bounded to a maximum of one.

### 2.4 Dynamic Deadlock Verification for Phasers

We conclude this section by outlining the phaser-based approach of Armus to barrier-deadlock verification. The challenge of deadlock verification for phasers is that tasks may participate in synchronisations on a phaser only at selected phases. Section 2.2 outlined WFG-based deadlock detection for basic barriers (i.e., barriers that require all member tasks must synchronise on every step). Considered simply, such approaches are unsuitable for phasers because they are based on capturing inter-task control flow dependencies at the granularity of barriers as synchronisation resources.

![Task-Event Graph](image)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 // t₁</td>
<td>1 // t₂</td>
<td>1 // t₃</td>
</tr>
<tr>
<td>2 arrv(a);</td>
<td>2 arrv(a);</td>
<td>2 arrv(a); ←</td>
</tr>
<tr>
<td>3 arrv(a);</td>
<td>3 arrv(a);</td>
<td>3 arrv(a);</td>
</tr>
<tr>
<td>4 await(a); ← 4 arrv(b);</td>
<td>4 arrv(b);</td>
<td></td>
</tr>
<tr>
<td>5 arrv(b);</td>
<td>5 await(b); ← 5</td>
<td></td>
</tr>
<tr>
<td>6 await(b);</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

In the above deadlock-free phaser code, all three tasks, t₁, t₂ and t₃, are registered to both phasers, a and b, and execution has reach the state indicated in each task: t₁ and t₂ are blocked, but not t₃. Naïve application of a basic WFG-based approach (i.e., building task-to-task wait-for dependencies by treating phasers as standard cyclic barriers) to this system would result in a false positive: a cycle arises because t₁ is blocked on a which t₂ is a member of, and t₂ is blocked on b which t₁ is a member of. The problem is that the construction of this false cycle involving two tasks and two phasers is insensitive to the fact that there are actually three phases in play.
On the other hand, bad asynchronous phase advancing patterns can easily give rise to deadlocks, as illustrated in Figure 2 (a). Both $t_4$ and $t_5$ will await on $a$, then await on $b$. However, $t_5$ blocks on its first phase of $b$ after arriving at its first phase on $a$, but without arriving at its second or third phases on $a$—which $t_4$ requires in order to progress to the arrive at its first phase on $b$.

**Armus: Task-Event Graphs.** Our approach is based on modelling the concurrency constraints of phaser systems at the granularity of *phases*. The intuition is that the operations performed by tasks on any given phaser induces an ordered series of *phase synchronisation events* on that phaser. (We may refer to phase synchronisation events as *phase events*, or simply *events*.) More specifically, a task may be related to a phase event because it is *waiting on* the event, *i.e.*, to participate in the synchronisation on this phase; or because the event is being *impeded by* the task, *i.e.*, the synchronisation event cannot occur because the task has not reached this phase.

Armus introduces a **Task-Event Graph** (TEG) model that can be considered as adapting traditional Task-Resource Graph models [61] to phaser systems by treating each phase event as a distinct temporal synchronisation resource. The resulting model is a bipartite graph of task and event nodes with the following key characteristics:

- A TEG models separate task *wait-on* event, and event *impede-by* task, relations.
- The impede-by relation incorporates a notion of *phase-transitivity* induced by event ordering: an event is inherently impeded by a task, if the task is associated with another event at any *earlier* phase on the same phaser.

As a preliminary example, Figure 2 (b) depicts the TEG for the deadlock situation of Figure 2 (a). The notation, *e.g.*, $a^3$ means phase 3 on phaser $a$. The solid edges $(t_4, a^3)$ and $(t_5, b^1)$ are given by the wait-on relation, and $(a^3, t_5)$ and the dashed edges $(b^1, t_4)$ by the impeded-by relation. We can explain the $(a^3, t_5)$ edge by because (i) there is a task waiting on $a^3$ (in this case $t_4$), and (ii) task $t_5$ is waiting on $b^1$ while having only reached a phase that precedes $a^3$ (in this case $t_5$ reached $a^1$).

In Sections 3 to 5, we formalise the Armus phaser language and dynamic deadlock verification methodology, and show that the verification is sound and complete. Section 6 discusses our implementations of Armus for X10 and Java.

### 3 A CORE PHASER-BASED LANGUAGE FOR GENERAL BARRIER SYNCHRONISATIONS

This section introduces the syntax and semantics of a core concurrent language with phasers, which we refer to as **Brenner**. The language is designed to express abstractions of concurrent, imperative barrier programs, sufficient to formalise our deadlock verification and show the verification properties. The main purposes of the formalism are to define the information required of a phaser system to characterise a deadlock, namely the state of the data structure underlying each phaser and the set of blocked tasks, and to model how the phaser operations act on this information. Since the run-time verification approach of Armus works by sampling the state of phasers and blocked tasks during program execution, the correctness of the deadlock analysis is independent of control flow mechanisms. Language constructs that do not directly affect barrier synchronisation are omitted, such as local data operations, or simplified, *e.g.*, looping constructs are abstracted as a non-deterministic loop statement.

#### 3.1 Brenner: A Core Phaser-Based Language

*Phasers.* We first formalise the core functionality of phasers. Let $P$ denote a *phaser* that maps *task names* $t, t', \ldots \in T$ to *local phases*, ranging over the natural numbers, $n \in \mathbb{N}$. Predicate $\text{await}(P, n)$, used by tasks to observe a phase event, holds iff every member of the phaser has a local phase of at
Dynamic Deadlock Verification for General Barrier Synchronisation

Phasers

\[ \exists t': P(t') \leq n \]

\[ P \xrightarrow{\text{reg}(t,n)} P \cup \{ t: n \} \]  \hspace{1cm} \text{[add]}

\[ P \cup \{ t: n \} \xrightarrow{\text{dereg}(t)} P \]  \hspace{1cm} \text{[drop]}

\[ P \cup \{ t: n \} \xrightarrow{\text{arrv}(t)} P \cup \{ t: n + 1 \} \]  \hspace{1cm} \text{[adv]}

Phaser maps

\[ M \xrightarrow{P=p} M \cup \{ p: P \} \]  \hspace{1cm} \text{[add-p]}

\[ P \xrightarrow{\phi} P' \]  \hspace{1cm} \text{[upd]}

Instructions

\[ \text{skip}; s \rightarrow s \]  \hspace{1cm} \text{[skip]}

\[ s' = c_1; \ldots; c_n; \text{end} \]

\[ \text{loop } s'; s \rightarrow c_1; \ldots; c_n; (\text{loop } s'; s) \]  \hspace{1cm} \text{[i-loop]}

\[ \text{loop } s'; s \rightarrow s \]  \hspace{1cm} \text{[e-loop]}

States

\[ t'' \notin \text{fv}(s) \]

\[ (M, T \cup \{ t': t = \text{newTid}(); s \}) \rightarrow (M, T \cup \{ t': s[t''/t] \} \cup \{ t'': \text{end} \}) \]  \hspace{1cm} \text{[new-t]}

\[ (M, T \cup \{ t': \text{fork}(t); s' \} \cup \{ t: \text{end} \}) \rightarrow (M, T \cup \{ t': s' \} \cup \{ t: s \}) \]  \hspace{1cm} \text{[fork]}

\[ M \xrightarrow{q=p} M' \]  \hspace{1cm} \text{[new-p]}

\[ P = \{ t: 0 \} \]  \hspace{1cm} \text{[skip]}

\[ q \notin \text{fv}(s) \]

\[ (M, T \cup \{ t: p = \text{newPhaser}(); s \}) \rightarrow (M', T \cup \{ t: s[q/p] \}) \]

\[ M(p)(t') = n \]

\[ (M, T \cup \{ t: \text{reg}(t,p); s \}) \rightarrow (M', T \cup \{ t': s \}) \]  \hspace{1cm} \text{[reg]}

\[ M \xrightarrow{p,\text{dereg}(t)} M' \]

\[ (M, T \cup \{ t: \text{dereg}(p); s \}) \rightarrow (M', T \cup \{ t: s \}) \]  \hspace{1cm} \text{[dereg]}

\[ M \xrightarrow{p,\text{arrv}(t)} M' \]

\[ (M, T \cup \{ t: \text{arrv}(p); s \}) \rightarrow (M', T \cup \{ t: s \}) \]  \hspace{1cm} \text{[arrv]}

\[ M(p) = P \]

\[ (M, T \cup \{ t: \text{await}(p,n); s \}) \rightarrow (M, T \cup \{ t: s \}) \]  \hspace{1cm} \text{[await-n]}

\[ M(p)(t) = n \]

\[ (M, T \cup \{ t: \text{await}(p,n); s \}) \rightarrow (M, T') \]  \hspace{1cm} \text{[await]}

\[ \text{s} \rightarrow \text{s'} \]

\[ (M, T \cup \{ t: s \}) \rightarrow (M, T \cup \{ t: s' \}) \]  \hspace{1cm} \text{[c-flow]}

Fig. 3. Operational semantics of BRENNER.
Listing 3. The Brenner representation of the X10 example in Listing 1.

```java
// Body of the main (parent) task t0:

pc = newPhaser(); // X10 clock (cyclic barrier), parent task implicitly registered
pf = newPhaser(); // X10 finish (join barrier), parent task implicitly registered

// Child task

loop
  t = newTid(); // Child task
  reg(pf, t); reg(pc, t); // X10: (implicit) finish reg, (explicit) clock reg
  fork(t)
    loop
      skip;
      arrv(pc); await(pc); // clock advance ×
      skip;
      arrv(pc); await(pc); // clock advance
      end;
  dereg(pc); // X10 task termination..
  dereg(pf); // ..deregisters the task from all barriers
  end;
  end;

arrv(pf); await(pf); // finish synchronisation ×

dereg(pf); // finish statement exit
end

least n:

\[
\text{await}(P, n) \triangleq \forall t \in \text{dom}(P): P(t) \geq n
\]

For a map \(X\), we write \(\text{dom}(X)\) for the domain of \(X\), and \(\text{img}(X)\) for the image of \(X\). When \(X \cap Y = \emptyset\) for some map \(Y\), we write \(X \uplus Y\) for the disjoint union of \(X\) and \(Y\).

Three atomic operations \(\phi\) mutate a phaser, as defined by “Phasers” in Figure 3. \(\text{reg}(t, n)\) registers a task named \(t\) to phaser \(P\) with initial phase \(n\), provided that the task is not already a member. \(\text{dereg}(t)\) removes the calling task \(t\) from the membership of \(P\). \(\text{arrv}(t)\) increments the local phase of \(t\) in \(P\).

Let a phaser map \(M\) be a map from phasers names \(p, p', \ldots \in \mathcal{P}\) to phasers, used to record all the phasers in a system. There are two operators \(o\) on phaser maps: \(p := P\) names the new phaser \(P\) with a global name \(p\), and \(p.\phi\) updates the phaser named \(p\) according to \(\phi\).

Syntax. Brenner abstracts a user-level program as a sequence \(s\) of instructions \(c\), generated by the grammar:

\[
s ::= c; s | \text{end}
\]

\[
c ::= \quad t = \text{newTid}() \mid \text{fork}(t) s \mid \text{loop} s \mid \text{skip} \\
\quad \mid p = \text{newPhaser}() \mid \text{reg}(t, p) \mid \text{dereg}(p) \mid \text{arrv}(p) \mid \text{await}(p) \mid \text{await}(p, n)
\]

Operational semantics. The reduction of Brenner terms is defined by “Instructions” and “States” in Figure 3. A task map \(T\) maps task names \(t_i\) to instruction sequences \(s_i\), representing the current state of the running tasks. A system state, or simply state, is a pair \(S::=(M, T)\).

We explain the syntax and operational semantics through Listing 3, which gives the Brenner representation of the X10 example from Listing 1. Spawning a new task comprises two instructions:
create a fresh task name bound as \( t \) by `newTid` (e.g., Line 5), and fork a task with this name to perform instruction sequence \( s \) by `fork(t)` \( s \) (e.g., Lines 7 to 16). The former adds a dummy (non-executable) task as a placeholder in the task map, to reserve the name \( t \) until the latter occurs.

Regarding task membership, `newPhaser` creates a phaser and registers the current task at phase zero. Rule `[reg]`, with `reg(p, t)`, lets some task \( t' \) register a new task \( t \) with phaser \( p \). Task \( t' \) must be registered with \( p \), and \( t \) inherits the phase of \( t' \);\(^1^{11}\) rule `[add]` enforces that \( t \) is not already a member of \( p \). Additionally, condition \( P(t') \leq n \) guarantees that there is some task, trivially the caller \( t' \), that is registered on phase \( n \), so as to ensure that phaser synchronisation is deterministic—observing a phase must be a stable property; otherwise, there would be no way to know when synchronisation happened, as new participants could be introduced in past phases. Operation `dereg(p)` deregisters the current task from phaser \( p \). In the example, the parent task creates a phaser representing the X10 join barrier \( p_f \) in Line 3, and registers child tasks \( t \) to \( p_f \) in Line 6, which deregister from \( p_f \) to signal task termination in Line 15.

For synchronisation, `arrv(p)` is the non-blocking operation for the current task to arrive at its next local phase, and `await(p)` blocks the current task until `await(P, n)`, where \( n \) is the local phase of \( t \) on \( P \), the phaser named \( p \). In the inner loop of the example (Lines 10 to 12), each child task advances its phase and then awaits the others to do the same. The variant `await(p, n)` takes \( n \) as an explicit argument and does not require \( t \) to be registered to \( p \), which captures use cases such as the `wait-only` phaser "registration" mode of HJ [62].\(^1^{12}\)

Lastly, the structural rule for control flow is standard. In BRENNER, local data operations are abstracted as `skip`, and the non-deterministic `loop`, which unfolds its body an arbitrary number of times (possibly zero), is used to subsume the control flow of standard conditional branches, while-loops and so on.

With respect to the dynamic deadlock verification, rules `[await]` and `[await-n]` are used to define the notion of blocked tasks, in order to characterise deadlocked states (Section 3.2) and establish the results in Section 5. Second, the operational semantics as whole serves as a specification of how phaser system state should be maintained by an implementation of ARMUS verification (or conversely, a specification of the phaser systems to which an implementation of ARMUS applies), as we discuss for X10 and Java in Section 6.2.

### 3.2 Phaser Deadlocks

A `phase event` \( e \), or simply `event`, is a pair \((p, n)\), which may be written as \( p^n \). The ordering of phase events on a phaser is given by the `precedes` relation on events, \( e < e' \):

\[
\frac{n < m}{(p, n) < (p, m)}
\]

Given a state \( S = (M, T) \), a task \( t \in T \) and an event \( e = (p, n), p \in \text{dom}(M), \) we define:

- \( t \) is `waiting on` \( e \), notation \( t \text{ WAIT-ON}_S e \), iff \( t \) is awaiting phase \( n \) on \( p \). That is, there exists \( s \) such that \( T(t) = \text{await}(p, n); s \), or \( T(t) = \text{await}(p); s \) and \( M(p)(t) = n \). In such cases, we also simply say \( t \) is `awaiting`.
- \( t \) is `associated with` \( e \), notation \( t \text{ ASSOC}_S e \), iff \( M(p)(t) = n \).
- \( e \) is `impeded by` \( t \), notation \( e \text{ IMPEDE-BY}_S t \), iff \( e \text{ ASSOC}_S e' \) such that \( e' < e \) and there exists a task \( t' \) where \( t' \text{ WAIT-ON}_S e \). In such a case, we also simply say \( e \) is `impeded`.

Given an \( S \), we will write simply `WAIT-ON_S` to denote the set of all \((t, e)\) pairs such that \( t \text{ WAIT-ON}_S e \); similarly for `IMPEDE-BY_S`.

\(^{11}\)Phase inheritance subsumes the X10 notion of child tasks inheriting the "initiation status" of split-phase synchronisations.

\(^{12}\)In HJ, `wait-only` registration is a special case where the task is implicitly assigned a local phase of \( \infty \) and is not permitted to arrive on the phaser. In ARMUS, this is modelled as allowing non-member tasks to await (but not arrive) on a phaser.
An \( e \) impeded by \( t \) relationship, where \( e = (p, n) \), signifies that \( t \) has a strictly earlier local phase on \( p \) than \( n \), and is thus required to perform some action, namely either at least one \( \text{arrv}(p) \) or a \( \text{dereg}(p) \), before \( e \) can be successfully observed by any awaiting task. Moreover, there is at least one such task \( t' \) awaiting \( e \), which ensures that the impede-by relation is finite. Note that, while a task may be waiting on at most one event, an event may be impeded by multiple tasks.

Unlike the wait-for relation in WFG-oriented approaches, neither wait-on nor impede-by inherently capture any notion of a task being stuck in of themselves. Instead, we naturally characterise phaser deadlock based on mutual dependencies between the two relations as follows. We define deadlocked states (i.e., local deadlock, as discussed in Section 2.2) based on totally deadlocked states (global deadlock). A totally deadlocked state occurs when every task is waiting-on some event and the event is impeded by some task.

**Definition 3.1 (Totally deadlocked state).** A state \((M, T)\) is totally deadlocked iff \( T \neq \emptyset \) and \( \forall t \in \text{dom}(T). \exists e \in \text{dom}(M) \times \mathcal{N}.(t \text{ wait-on}_S e \land e \text{ is impeded}) \).

A totally deadlocked state extended with tasks that are not awaiting impeded events is considered as simply deadlocked, as the system may still potentially progress by the reduction of these additional tasks.

**Definition 3.2 (Deadlocked state).** A state \( S = (M, T \cup T') \) is deadlocked on \( T \) iff the state \((M, T)\) is totally deadlocked. In such a case, we also simply say \( S \) is deadlocked.

The notion of local deadlocks is crucial for applications that may never terminate (typical examples being operating systems and persistent network services), and practically important for systems that may simply be long running.

### 4 Dynamic Deadlock Verification for Phasers

This section first defines the construction of Task-Event Graphs (TEGs) from Brenner system states. Second, by starting from a more general model oriented to TEGs (as opposed to a directly WFG-oriented approach), we are able to recover smaller but equivalent representations as optimisations, with respect to the verification properties (Section 5). These are the WFG, and the counterpart notion of State Graph (SG).

#### 4.1 Task-Event Graphs

**Background.** A Task-Resource Graph (TRG) [61], also known as Transaction-Resource Graph, is a bipartite directed graph used to model concurrency constraints between tasks and resources. We adapt this term as Task-Event Graph (TEG) in Armus, since we model the concurrency constraints arising from tasks observing transient phase events by collective synchronisation operations, as opposed to individual acquisition and release actions on “concrete” resources.

Holt generalised TRGs to General Resource Graphs (GRG) [35], by augmenting resource-nodes with the number of available resources. Unlike TEGs, a GRG cycle does not necessarily imply deadlock. The GRG must first be transformed a finite number of steps to identify a potential deadlock. Non-bipartite directed graphs of tasks and synchronisation mechanisms have also been used to detect lock-based deadlocks [52].

Coffman et al. introduced the State Graph (SG) [12] to model concurrency constraints directly between synchronisation mechanisms. State-of-the-art on identifying potential lock-based deadlocks includes approaches based on SGs [7, 23, 58], where SGs also known as Lock-Order Graphs and Lock-Dependency Graphs. SGs have also been used to infer deadlock-free contracts for concurrency libraries [21].
Knapp introduced the *Wait-For Graph* (WFGs) \[40\] to model concurrency constraints directly between tasks. As discussed earlier, variations of WFGs are used in the state-of-the-art on deadlock detection for distributed message passing and (static membership) barriers \[34, 37\].

**Basic concepts from graph theory.** A (directed) graph $G$ is a pair $(V, A)$ comprising a nonempty finite set of vertices $V$, ranged over by $v, u$, and a finite set of arcs $A$, ranged over by $a, b, c$, where an arc $a$ is a pair $(v, u)$ with $v, u \in V$. An arc $(v, u)$ is directed from its head $v$ to its tail $u$. We write $a \in G$ to mean $G = (V, A)$ and $a \in A$. Graph $(U, B)$ is a subgraph of graph $(V, A)$ iff $U \subseteq V$ and $B \subseteq A$.

A walk $w$ on $(V, A)$ is a (possibly empty) sequence $a_1 \cdots a_n$ (also written $a_1, \ldots, a_n$) of arcs in $A$ such that, for all $i < n$, $a_i = (v_i, v_{i+1})$ and $a_{i+1} = (v_{i+1}, v_{i+2})$. We write $e$ to denote the walk of length zero and $a : w$ to preprend edge $a$ to walk $w$. For instance, walk $w_1 : (v_1, v_2) : (v_2, v_3) : \epsilon$ is an alternative notation for walk $(v_1, v_2) \cdot (v_2, v_3)$.

We write $w = a_1 \cdots a_n$, where there exists $i \in \{1, \ldots, n\}$ and $a = a_i$; and $v \in w$ to mean there exists $(v_1, v_2) \in w$ and $v \in \{v_1, v_2\}$. We have that $v_2 \in (v_1, v_2) \cdot (v_2, v_3)$ and that $(v_2, v_3) \in (v_1, v_2) \cdot (v_2, v_3)$, yet $v_4 \not\in (v_1, v_2) \cdot (v_2, v_3)$ and that $(v_5, v_3) \not\in (v_1, v_2) \cdot (v_2, v_3)$.

We may write a walk by its constituent vertices, i.e., $(v_1, v_2) \cdot (v_2, v_3) \cdots (v_{n-1}, v_n)$ abbreviated as $v_1 : v_2 : \cdots : v_{n-1} : v_n$. For instance, walk $v_1 : v_2 : v_3$ is an alternative notation for walk $(v_1, v_2) \cdot (v_2, v_3)$.

We may also simply refer to a walk by its first and last vertices, i.e., a $v_1 : v_n$ walk means a nonempty walk $v_1 \cdots v_n$. Given a walk $w = a_1 \cdots a_n$ such that $n \geq 1$, we have that $a_n$ is the last arc of walk $w$. A cycle is a $v$-$v$ walk—note that cycles are nonempty walks.

A bipartite graph $G = (V, U, A)$ is a graph $(V \cup U, A)$ where $V$ and $U$ are disjoint and, for all $a \in A$, $a = (v, u)$ or $a = (u, v)$ with $v \in V$ and $u \in U$.

**Task-Event Graphs.** A Task-Event Graph (TEG) is a bipartite graph where the two disjoint sets of vertices are task names $t \in T$ and events $e \in P \times N$. A TEG thus has two kinds of arcs: wait-on arcs $(t, e)$ directed from a task $t$ to an event $e$, and impede-by arcs $(e, t)$ from an event $e$ to a task $t$.

**Definition 4.1 (Associated TEG).** Given a state $S$, let $W = \text{WAIT-ON}_S$ and $I = \text{IMEPEDE-BY}_S$. The TEG associated with $S$, $\text{teg}(S)$, is the bipartite graph $(U, U', A)$, where $U = \text{dom}(W) \cup \text{img}(I)$, $U' = \text{img}(I) \cup \text{dom}(W)$ and $A = W \cup I$.

**Example 4.1.** Figure 4 involves three tasks and two phasers. Execution (necessarily) reaches the totally deadlocked state $S$ indicated in the code, where (assuming the terminated main task named $t_0$):

$S = (M, T) = \{ p : \{ t_1 : 2, t_2 : 0, t_3 : 1 \}, q : \{ t_1 : 0, t_2 : 1 \} \}$

$T = \{ t_0 : \text{end}; t_1 : \text{await}(p); \text{arrv}(q); \text{await}(q); \text{end};$

$t_2 : \text{await}(q); \text{arrv}(p); \text{await}(p); \text{end}; t_3 : \text{await}(p); \text{arrv}(p); \text{await}(p); \text{end} \}$

Then $\text{teg}(S) = \{ (t_1, t_2, t_3), (p^1, p^2, q^1) \}$, $\text{WAIT-ON}_S \cup \text{IMEPEDE-BY}_S$, where:

$\text{WAIT-ON}_S = \{ (t_1, p^2), (t_2, q^1), (t_3, p^1) \}$ (Solid edges)

$\text{ASSOC}_S = \{ (t_1, p^2), (t_1, q^0), (t_2, p^0), (t_2, q^1), (t_3, p^1) \}$

$\text{IMEPEDE-BY}_S = \{ (p^1, t_2), (q^1, t_1), (p^2, t_2), (p^2, t_3) \}$ (Dashed edges)

Note that this example features two deadlock cycles. Although there are only two phasers, their usage by the tasks induces three distinct events, leading to the cycle $t_1 \cdot p^2 \cdot t_3 \cdot p^1 \cdot t_2 \cdot q^1 \cdot t_1$. The smaller cycle, $t_1 \cdot p^2 \cdot t_2 \cdot q^1 \cdot t_1$, corresponds to a local deadlock in an intermediate situation where $t_1$ and $t_2$ have blocked but $t_3$ has not. The two cycles arise from $t_2$ impeding both $p$ events under observation: $t_2$ is associated with $p^0$, thus impeding $p^1$ and $p^2$. 

ACM Transactions on Programming Languages and Systems, Vol. 1, No. 1, Article 1. Publication date: January 2018.
1:18 Tiago Cogumbreiro, Raymond Hu, Francisco Martins, and Nobuko Yoshida

p = newPhaser(); q = newPhaser();
t1 = newTid(); reg(t1, p); reg(t1, q);
t2 = newTid(); reg(t2, p); reg(t2, q);
t3 = newTid(); reg(t3, p);
fork(t1) arrv(p); arrv(p); await(p); // p\textsuperscript{2} ×
   arrv(q); await(q); // q\textsuperscript{1} end;
fork(t2) arrv(q); await(q); // q\textsuperscript{1} ×
   arrv(p); await(p); // p\textsuperscript{1} end;
fork(t3) arrv(p); await(p); // p\textsuperscript{1} ×
   arrv(p); await(p); // p\textsuperscript{2} end;
dereg(p); dereg(q);
end

(a) phaser code (deadlock state as indicated) (b) TEG

Fig. 4. A (totally) deadlocked phaser program (Example 4.1).

Example 4.2. Figure 5 demonstrates a deadlock due to a single task awaiting a future phase (i.e., a phase ahead of its local phase) on a phaser that it is registered to. In such cases, the relevant impede-by dependency is inherent from the impeding task being the same as the observing task: t\textsubscript{1} is associated with p\textsuperscript{0}, impeding p\textsuperscript{1} which t\textsubscript{1} is itself waiting on. This (anti-)pattern thus causes deadlock in any system context.

The feature of awaiting “future” phases can be used by unregistered tasks (e.g., wait-only consumers), for which this is technically the same as awaiting any arbitrary phase (since such tasks do not actually have a local phase on the phaser). Then there is no issue of deadlock because the associated-events predicate does not hold between such tasks and the observed events.

4.2 Deriving Wait-For Graphs and State Graphs from Task-Event Graphs
We can compress a bipartite TEG to a smaller model by vertex contraction: contracting the event vertices results in a Wait-for Graph (WFG), and contracting the task vertices results in a State Graph (SG).

Definition 4.2 (Associated WFG, SG). Assume a state S and its associated TEG, teg(S) = (U, U', A).
The WFG associated with $S$, $\wfg(S)$, is the graph $(U, A')$ where $A' = \{ (t, t') | \exists e \in U'. (t, e) \in A \land (e, t') \in A \}$.

The SG associated with $S$, $\sg(S)$, is the graph $(U', A')$ where $A' = \{ (e, e') | \exists t \in U. (e, t) \in A \land (t, e') \in A \}$.

Example 4.3. Consider the following deadlocked state $S = (M, T)$ of the running example from Listings 1 and 3, taking $\text{num\_tasks}$ to be 3. Tasks $t_1$, $t_2$, and $t_3$ are the child tasks waiting on the cyclic barrier (clock) $p_c$, and the main (parent) task $t_0$ is waiting on the join barrier (finish) $p_f$.

$M = \{ p_c: \{ t_1: 1, t_2: 0, t_3: 0, t_0: 0 \}, \ p_f: \{ t_1: 0, t_2: 0, t_3: 0, t_0: 1 \} \},$

$T = \{ t_0: \text{await}(p_f); s_0, \ t_1: \text{await}(p_c); s_1, \ t_2: \text{await}(p_c); s_2, \ t_3: \text{await}(p_c); s_3 \}$

(We omit the continuations $s_0..4$ for brevity.) Consequently, $\text{wait\_on}_S$ and $\text{impede\_by}_S$ are, respectively:

$$\{ (t_0, p_f^1), (t_1, p_f^1), (t_2, p_f^1), (t_3, p_f^1) \} \quad \{ (p_c^1, t_0), (p_f^1, t_1), (p_f^1, t_2), (p_f^1, t_3) \}$$

Figure 6 depicts the TEG associated with $S$, and by contraction the associated WFG and SG.

**Dynamic graph model selection.** A benefit of our approach is that the deadlock detection can be optimised by dynamically selecting between a WFG or SG model as appropriate. The selection can be guided by the cost of cycle detection. The following is the worst-case time complexity for using the WFG and the SG.

**Proposition 4.4 (time complexity).** Given a state $S$, let $W$ stand for $\text{wait\_on}_S$ and $I$ stand for $\text{impede\_by}_S$. Deadlock detection using the associated WFG is $O(|W|^2 + |W|)$, while deadlock detection using the SG associated with is $O(|I|^2 + |I|)$.

**Proof.** Cycle detection in a graph $(V, A)$ has a time complexity of $O(|A| + |V|)$ [66]. For any graph, $|A| \leq |V|^2$ [4], thus we can bound the complexity by $O(|V|^2 + |V|)$. Since the WFG vertices are the tasks, deadlock detection using the WFG with $|W|$ tasks has a complexity of $O(|W|^2 + |W|)$. Similarly for the SG, we have $O(|I|^2 + |I|)$. 

Based on this observation, we can expect the WFG or the SG to be more efficient than the other based on the ratio of tasks to synchronisation events in the run-time system. Some of the scenarios for each case are as follows.

WFGs are suitable when events outnumber tasks, which we may expect in situations where barriers are used to represent resources and the means to regulate their access. Such situations arise in dataflow/stream processing [64, 64, 71], and applications such as clocked variables [2]. This characteristic is further pronounced in applications of asynchronous phase advancing that relate.
events to resources, as in producer-consumer patterns, where a potentially large number of events may arise from even a small number of phasers and tasks.

By contrast, we may expect SGs to be suitable, due to tasks outnumbering events, in settings where parallelism is based on the scaling of tasks, such as single program, multiple data (SPMD) systems, e.g., using MPI or OpenMP, and phaser accumulators [63].

Ultimately, we expect that the ratio of tasks to events may be difficult to predict in many barrier applications, with the potential for significant variance during execution, motivating an approach to dynamic graph model selection. Such situations may arise in advanced languages such as X10, that support ad hoc combined use of multiple forms of barrier abstractions, and other hybrid systems, such as combinations of MPI and OpenMP.

Section 7.3 evaluates the performance impact of dynamic graph model selection in practice. The results – especially those for deadlock avoidance – confirm the above remarks, namely that the WFG is indeed more efficient in programs where events outnumber tasks, and vice versa for SG. We also note that in every benchmark application except for the simplest one (named SE), using the TEG directly (as a base case comparison) is always slower than using the best option out of WFG or SG.

5 DEADLOCK VERIFICATION PROPERTIES

This section presents correctness properties of phaser deadlock verification in Armus. First, we show that the WFG and the SG associated with a state $S$ are equivalent with respect to the presence of cycles. Second, we show that deadlock detection for a state $S$ by cycle detection in the associated WFG is sound and complete, i.e., the WFG contains a cycle if, and only if, $S$ is deadlocked. The definitions and proofs are available as a machine-checked Coq implementation.

5.1 Model-Equivalence Theorem

In this section we show that a state $S$, whenever there is a cycle in the associated WFG, there is also a cycle in the associated SG, and vice versa. Precisely, given a state $S$ and a cycle in the associated WFG, from $v_1$ to $v_1$, we can construct a path in the associated SG from $u_1$ to $u_n$, as depicted in the next graph, where the dotted arcs are in the WFG, the solid and the dashed arcs are both in the TEG, and the squiggly arcs are in the SG. From the edges $(v_1, u_1)$ and $(u_n, v_1)$, we show that the edge $(u_n, u_1)$, not depicted below, is an edge in the SG, and therefore there is a cycle in the SG that passes through $u_1$.

The equivalence of finding a cycle in the WFG and SG can be stated generally for any bipartite graph.

**Definition 5.1 (Contracted graph).** Let $G = (V, U, A)$ be a bipartite digraph. Let the contraction of $G$ into $G_V = (V, A_V)$ where $A_V$ is defined as $(v_1, v_2) \in A_V$ if, and only if, $\exists u \in U$ such that $(v_1, u) \in A$ and $(u, v_2) \in A$. Similarly, let the contraction of $G$ into $G_U = (U, A_U)$ where $A_U$ is defined as $(u_1, u_2) \in A_U$ if, and only if, $\exists v \in V$ such that $(u_1, v) \in A$ and $(v, u_2) \in A$. We call $a \in G_V$ a $V$-arc, and $a \in G_U$ a $U$-arc. Similarly, we call $w \in G_V$ a $V$-walk, and $w \in G_U$ a $U$-walk.

https://gitlab.com/cogumbreiro/brenner-coq
Henceforth until the end of the section, let \( v \) denote a vertex such that \( v \in V \) and let \( u \) denote a vertex such that \( u \in U \). In a bipartite graph’s contraction \( G_V \), any path with three vertices can be “translated” into a path in \( G_U \) that only has two vertices (i.e., a \( U \)-arc).

**Definition 5.2.** Let \((v_1, v_2) \sim (v_2, v_3) \sim_G (u_1, u_2)\) hold if the arcs \((v_1, u_1), (u_1, v_2), (v_2, u_2)\), and \((u_2, v_3)\) are all in \( G \).

It follows trivially that \((v_1, v_2), (v_2, v_3)\) are arcs in \( G_V \) and that \((u_1, u_2)\) is an arc in \( G_U \). The next graph depicts proposition \((v_1, v_2) \sim (v_2, v_3) \sim_G (u_1, u_2)\).

Now let \( w \sim_G w' \) relate a \( V \)-walk \( w \) with a \( U \)-walk \( w' \).

**Definition 5.3 (Walk contraction).** Let \( G = (V, U, A) \) be a bipartite digraph. Let \( w \sim_G w' \) be defined inductively as:

\[
\begin{align*}
\epsilon & \sim_G \epsilon \\
 a & \sim_G \epsilon \\
 a_1 \sim a_2 \sim_G b & \sim_G \epsilon \\
 (a_1::a_2::w_1) & \sim_G \epsilon \\
 (a_1::a_2::w_1) & \sim_G (b::w_2)
\end{align*}
\]

**Lemma 5.4.** Let \( G = (V, U, A) \) be a bipartite graph. If \( w \) is a \( V \)-walk, then there exists a \( U \)-walk \( w' \) such that \( w \sim_G w' \).

**Proof.** The proof follows by induction on the structure of \( w \). There are two cases to consider. The first case is when \( w = \epsilon \); the proof follows trivially taking \( \mathrm{w}' = \epsilon \). By definition we have that \( \epsilon \) is a \( U \)-walk and that \( \epsilon \sim_G \epsilon \) holds. The second case is when \( w = a::w_1 \) and we want to show that there is some \( U \)-walk \( w' \) such that \( a::w_1 \sim_G w' \). By inspecting the structure of \( w_1 \), we have two further sub-cases to analyse: (a) either \( w_1 = \epsilon \), or (b) \( a = (v_1, v_2) \) and \( w_1 = (v_2, v_3)::w_2 \).

(a) We conclude the proof of this sub-case with \( \mathrm{w}' = \epsilon \), since by definition \( \epsilon \) is a \( U \)-walk and \( a::\epsilon \sim_G \epsilon \) holds.

(b) By applying the induction hypothesis to \( w_2 \) is a \( V \)-walk, we get that there is some \( U \)-walk \( w'' \) such that \((v_2, v_3)::w_2 \sim_G w'' \). The sub-case concludes by showing that given a \( V \)-walk \((v_1, v_2)::(v_2, v_3)::w_2 \), a \( U \)-walk \( w'' \), and \((v_2, v_3)::w_2 \sim_G w'' \), there exists some \( U \)-walk \( w' \) such that \((v_1, v_2)::(v_2, v_3)::w_2 \sim_G w' \) holds. The proof follows by inverting the proposition \((v_2, v_3)::w_2 \sim_G w''\), which can be proved from knowing that for every path \( V \)-walk \( v_1 \cdot v_2 \cdot v_3 \) there exists a \( U \)-walk \( u_1 \cdot u_2 \) such that \( v_1 \cdot v_2 \cdot v_3 \sim_G u_1 \cdot u_2 \).

(The Coq version of this lemma is named \_to\_b\_total, in aniceto-coq/src/Graphs/Cycle.v)

**Lemma 5.5.** Let \( G = (V, U, A) \) be a bipartite graph. If \((u_1, u_2)\) is the last arc of \( U \)-walk \( w' \) and \( w \sim_G w' \), then there exist a \( V \)-walk \( v_1 \cdot v_2 \cdot v_3 \) such that \( v_1 \cdot v_2 \cdot v_3 \sim_G u_1 \cdot u_2 \).

**Proof.** The proof follows by induction on the derivation tree of \( w \sim_G w' \). Inverting proposition \( w \sim_G w' \), we get three cases to consider: (i) \( w = w' = \epsilon \); (ii) \( w = (v_1, v_2) \) and \( w' = \epsilon \); and (iii) \( w = (v_1, v_2)::(v_2, v_3)::w_1, w' = a::w_2, \) and \((v_2, v_3)::w_2 \sim_G (u_1, u_2)::w_2 \).

We conclude cases (i) and (ii) with the same proof. By hypothesis \((u_1, u_2)\) is the last arc of \( w' \), thus \((u_1, u_2) \in w' \); however, \( w' = \epsilon \), and therefore \((u_1, u_2) \in \epsilon \), which cannot be by definition of arc membership.

As for case (iii), we inspect the structure of \( w_2 \) and get two further sub-cases: (a) \( w_2 = \epsilon \) and (b) \( w_2 = (u_2, u_3)::w_3 \).
(a) At this point \( w' = a : e \) and therefore \( a = (u_1, u_2) \). We can now conclude by applying the hypothesis \( (v_2, v_3) : w_1 \leadsto_G (u_1, u_2) : e \).

(b) Recall we want to show that there exist a \( V \)-walk \( v_1 \cdot v_2 \cdot v_3 \) such that \( v_1 \cdot v_2 \cdot v_3 \leadsto_G u_1 \cdot u_2 \).

Since we know that \( (u_1, v_2) \) is the last arc of \( a : (u_2, u_3) : w_3 \), then arc \( (u_1, v_2) \) is also the last of walk \( (u_2, u_3) : w_3 \). Thus, we apply the induction hypothesis to conclude our proof.

\( \square \)

**Lemma 5.6.** Let \( G = (V, U, A) \) be a bipartite graph. If the \( V \)-walk \( w \) is a cycle, then there exists a \( U \)-walk \( w' \) that is a cycle.

\[ \text{Proof.} \] Applying Lemma 5.4 to our hypothesis we get that there exists a \( U \)-walk \( w' \) such that \( w \leadsto_G w' \) holds. By inverting the latter there are two cases to consider: (a) \( w = (v, u) : e \); and (b) \( w = a : w_1 \) and \( w' = a_2 : w_2 \).

Case (a) there exists a vertex \( u \) such that we have \((v, u) \in A \) and \((u, v) \in A \). We conclude the proof since \((u, v) \) is a cycle in the \( U \)-graph.

Case (b) since, \( w \) is a cycle, let \( w = (v_1, v_2) : w_1 \) and \((v_n, v_1) \) be the last arc of \( w \), when \( a_1 = (v_1, v_2) \). Applying Lemma 5.5 to the hypothesis that \((v_n, v_1) \) is the last arc of \( w \), we get: \( v_{n-1} \cdot v_n \cdot v_1 \leadsto_G u_{n-1} \cdot u_n \). We illustrate the two paths bellow.

![Diagram showing two paths](image)

From \((u_n, v_1) \in A \) and \((v_1, u_1) \in A \) we get that \((u_n, u_1) \) is an edge in \( G_U \), thus

\[ \text{(The Coq version of this lemma is named cycle a to b, in aniceto-coq/src/Graphs/Cycle.v) \square} \]

**Corollary 5.7.** There exists a cycle \( w \) on graph \( \text{wfg}(S) \) if, and only if, there exists a cycle \( w' \) on graph \( \text{sg}(S) \).

\[ \text{Proof.} \] We apply Lemma 5.6 to each side of the implication.

\[ \text{(The Coq version of this corollary is named sg to wfg and wfg to sg, in brenner-coq/src/ResourceDependency.v)} \square \]

### 5.2 Soundness

The property of soundness ensures the absence of false positives, that is, soundness entails that if there is a cycle \( w \) in the WFG of a given state \( S \), then such state is deadlocked. The proof is split into two main steps. First, we divide the task map from state \( S \) into two disjoint task maps, according to the membership of the tasks (vertices) in cycle \( w \). Second, we then show that any state whose task map is composed of the vertices mentioned in cycle \( w \) is totally deadlocked, which allow us to conclude that state \( S \) is deadlocked.

**Lemma 5.8.** Let \( S = (M, T) \) and \( G \) be the WFG associated with \( S \). Let \( w \) be a walk on \( G \) such that \( t \in w \) if, and only if, \( t \in \text{dom}(T) \). If \( w \) is a cycle, then state \( S \) is totally deadlocked.
PROOF. To show that $S$ is totally deadlocked, we must prove that: (i) all tasks in $S$ are waiting on some event, (ii) all tasks are being impeded by some event, and (iii) $T$ is nonempty.

Part (i). We need to show that if $t \in \text{dom}(T)$, then there exists an event $e$ such that $t \text{ wait}-\text{ON}_S e$. Since $t \in \text{dom}(T)$, from the hypothesis we have that $t \in w$. Given that $w$ is a cycle and that $t \in w$, then there exists a task $t'$ such that $(t, t') \in w$, which we invert to conclude that $t \text{ wait}-\text{ON}_S e$.

Part (ii). We need to show that if $t \text{ wait}-\text{ON}_S e$, then there exists some task $t'$ such that $t \text{ impede-by}_S t'$. From $t \text{ wait}-\text{ON}_S e$, we get that $t \in \text{dom}(T)$ and by hypothesis $t \in w$. But as $w$ is a cycle and $t \in w$, then there exists some vertex $t'$ such that $(t, t') \in w$. From $(t, t')$ and $t$ is waiting on $e$, we have that $e \text{ impede-by}_S t'$.

Part (iii). Task map $T$ is nonempty, since $w$ is a cycle, which by definition has at least one vertex $t \in w$, thus, by hypothesis, $t \in \text{dom}(T)$.

(The Coq version of this lemma is named soundness_totally, in brenner-coq/src/Soundness.v)

5.3 Completeness

The property of completeness entails the absence of false negatives, that is, for any deadlocked state $S$ we can exhibit a cycle in the WFG of $S$. The proof is divided into two steps. First, we consider totally deadlocked states $S$, in which we observe that each task is a vertex in the WFG of $S$ with an outgoing arc. There is a cycle in any finite graph whose vertices have at least an outgoing arc, so totally deadlocked states have a cycle. Second, we show the WFG of a totally deadlocked state is a subgraph of the WFG of the relative deadlocked state, thus we can conclude our proof.

LEMMA 5.10. Let $G = (V, A)$ be the WFG associated with some state $S$. If $S$ is totally deadlocked, then there exists a cycle $w$ in $G$.

PROOF. We know that if (i) $A$ is nonempty, and (ii) all vertices in $G$ have an outgoing arc, then $G$ has a cycle. Our mechanisation provides a constructive proof of this result, yet since this is a standard result outside of the focus of this paper we redact its discussion.

Part (i), we show that graph $A$ is nonempty. Let $S = (M, T)$. Since $S$ is totally deadlocked, then there exists some $t \in \text{dom}(T)$. Furthermore, we know that all tasks are waiting on some event $e$, thus $t \text{ wait}-\text{ON}_S e$. But given that $S$ is totally deadlocked, then there is some task $t'$ such that $e \text{ impede-by}_S t'$. Hence, $(t, t') \in G$ and therefore $G$ is nonempty.

Part (ii), we show that if $t \in G$, then there exists some task $t'$ such that $(t, t') \in G$. From $t \in G$ and the definition of totally deadlocked, we get that there exists an event $e$ such that $t \text{ wait}-\text{ON}_S e$.
Hence, by definition of totally deadlocked, there exists some task $t'$ such that $e \text{ impede-by}_S t'$ and $t' \in S$; let us take $t'$. From $t \text{ wait-on}_S e$ and $e \text{ impede-by}_S t'$, we get that $(t, t') \in G$.

(The Coq version of this lemma is named totally_deadlock_has_cycle, in brenner-coq/src/Completeness.v)

\[\square\]

**Theorem 5.11.** If $S$ is deadlocked, then there exists a walk $w$ such that $w$ is a cycle in the WFG of state $S$.

**Proof.** Let graph $G$ be the WFG associated with $S$. Now, by inverting the hypothesis that $S$ is deadlocked we get that $S = (M, T \cup T')$, $(M, T)$ is totally deadlocked, and $G'$ is the WFG associated with $(M, T)$ such that $G'$ is a subgraph of $G$. From Lemma 5.10 and $(M, T)$ being totally deadlocked, we get that there exists a cycle $w$ in $G'$. Yet, since $G'$ is a subgraph of $G$, then $w$ is a cycle in $G$.

(The Coq version of this theorem is named completeness, in brenner-coq/src/Completeness.v)

\[\square\]

## 6 ARMUS IMPLEMENTATIONS

This section discusses the implementation of Armus for X10 and Java. Our open source implementations are the first sound and complete tools for barrier-deadlock verification in both cases. Key features are scalability from dynamic selection between WFG and SG models (Section 6.1) and support for distributed barriers (Section 6.3).

Armus is implemented as a two layer framework. The Verification Layer is a platform-independent core library for managing the monitored system state and performing the deadlock detection. The Application Layer consists of a specific implementation for each target language. It is responsible for correlating the barrier operations in the target language with Armus phaser operations, to extract and maintain the system state in a consistent manner.

### 6.1 Verification Layer

The Verification Layer (VL) is a Java library that has two main purposes: maintenance of the system state required by the deadlock verification; and the actual deadlock checking.

**Overall methodology.** Based on the formal developments in the preceding sections, we give a practical methodology for deadlock verification that is readily applicable to existing barrier and phaser systems, including distributed implementations. A key point that we leverage: Armus deadlock verification can be performed on a composition of per-task views of the system state obtained from only awaiting tasks. Such “partial” system views are safe abstractions of the centralised, global view represented by a formal system state $S$, w.r.t. the deadlock verification.

The methodology stipulates: whenever a task enters a potentially blocking await operation, the event it is waiting on and the set of events currently associated with that task are recorded; we refer to this localised information as the blocking status of the task. The blocking status recorded for a task is cleared on completion of the await operation. Note that a blocking status is invariant while the task remains awaiting; in particular, its phaser memberships.

The deadlock verification is then conducted as follows:

1. A snapshot of the global wait-on and associated-events relations (Section 3.2) is obtained by compiling all the currently recorded blocking statuses.
2. The impede-by relation is derived from the above, giving the base components of the TEG (Definition 4.1).
3. The TEG components are used to construct the associated WFG or SG (Definition 4.2). The system is deadlocked if, and only if, there is a cycle in the constructed graph.
Given the blocking statuses of all currently awaiting tasks, the obtained wait-on is the same as \text{wait-on}_S for the full system state \( S \), but the associated events relation is the subset of \text{assoc}_S restricted to awaiting tasks only. In comparison to the core definitions in Section 3.2, this restriction serves as a safe optimisation that reduces the size of derived impede-by. The restriction preserves soundness because new cycles are never introduced (edges may only be pruned), and completeness because existing cycles are always retained (all tasks involved in a cycle are in \text{wait-on}_S).

\textit{VL state management.} Following this methodology, the VL maintains Armus system state as a map from tasks to \textit{blocking status} records. Its key set comprises the tasks that are currently executing an operation corresponding to an Armus phaser await. Each record is a pair: the event that the task is waiting on, and the set of events associated with the task; this information is provided by the Application Layer (Section 6.2). The VL directly maintains the system state as per-task records (as opposed to the derived wait-for and impede-by dependencies) to optimise the processing of operations related to updating the blocking statuses, since they are more frequent than deadlock checking.

\textit{Graph selection and cycle checking.} The deadlock checker, following steps (1)–(3), implements the core functionality for: compiling the wait-for and impede-by dependencies from the blocking statuses, graph model selection and construction, and cycle detection. We use JGraphT\(^{14}\) to perform cycle detection.

The VL supports two graph selection modes: \textit{static} and \textit{dynamic}. In the static mode, the deadlock checker always uses the specified model type (\textit{cf.} inherent coupling to WFGs by design, \textit{e.g.}, [34]). In the dynamic mode, the graph model is dynamically selected according to the heuristics described below, meaning that the verification may switch between models during execution. We outline the implementation of each mode.

- The \textit{WFG-static} mode closely follows the main methodology (as outlined in Section 6.1) by constructing the associated WFG in two passes over the blocking status records. The first derives the impede-by dependencies. The second constructs the WFG by generating an edge from each event being waited on by a task to the events impeded by that task.
- In the \textit{SG-static} mode, SG construction is optimised into one pass by directly generating an edge from each event \( e \) associated with the task to the event the task is waited on, \textit{excluding} \( e \). This optimisation is possible because the barrier facilities of Java and X10 support only the \texttt{await}(\( p \)) form of awaiting (\textit{i.e.}, not \texttt{await}(\( p, n \))), and hence: (1) it is unnecessary to build the SG edges transitively; and (2) SG cycles of length one, \((e, e)\), cannot arise.

A trade-off of conducting a single pass is that the resulting graph contains at least the nodes of the formally defined SG, and possibly more, namely events modelled as being impeded but not being observed by any task. However, this remains correct because these additional nodes have no incoming edges (so no new cycles are introduced) and no edges are removed.

- The \textit{dynamic} mode starts with the SG construction pass, but additionally builds the impede-by dependencies alongside. We employ a heuristic during this pass: if the number of SG-edges exceeds the number of blocked status records processed by a configurable threshold, the VL switches to WFG construction (by finishing the building of impede-by). By default, our implementation uses a threshold of twice as many SG-edges as tasks, obtained from practical experiments. If the initial SG pass completes, the VL may still opt to build the WFG according to ratio of tasks to event nodes (following Proposition 4.4).

\(^{14}\)http://jgrapht.org/
6.2 Application Layer

We present implementations of the Application Layer (AL) for verifying barrier deadlocks in X10 and Java, Armus-X10 (Section 6.3) and JArmus (Section 6.4), respectively. The AL implementations serve as frontend user tools that work by "weaving" Armus verification instructions into the program. Armus-X10 currently supports the Java backend of X10 (Managed X10); an implementation for the C++ backend would follow the same principles.

Armus-X10 and JArmus are implemented as a post-compilation step, taking the generated Java bytecode as input. Any Java/X10 program that passes standard compilation is accepted by the Armus tools, returning a valid Java/X10 program (i.e., with respect to standard JVM dynamic class verification) that is modified only by the insertion of Armus instructions. The resulting program thus features the same barrier usage as the original, but with the Armus dynamic deadlock verification guarantees for the targeted barrier programming facilities (detailed below). Both implementations use AspectJ\footnote{https://eclipse.org/aspectj/} to weave the required VL calls around the target operations; e.g., to pass the blocking status to the VL on entering a potentially blocking barrier operation, and to clear the blocking status afterwards.

**Deadlock detection and avoidance.** Our implementations support deadlock avoidance in addition to standard detection. Avoidance mode is implemented by running the main verification methodology (as outlined above) inline with every invocation of a potentially blocking operation. This is achieved by weaving both the VL state update and deadlock checking calls around every such target operation. From the user perspective, the target operation is interrupted by an exception if it will introduce a deadlock. For certain applications, such exceptions may be handled by the programmer in a manner that promotes resilience to deadlocks. See Section 8 for further discussion of deadlock avoidance.

In the default detection mode, only the VL state update calls are weaved into the user program; Armus performs the deadlock verification periodically. In contrast to avoidance, the detection mode only reports already existing deadlocks, with lower performance overhead. The comparative performance of detection and avoidance is evaluated in Section 7.1.

6.3 Armus-X10

Armus-X10 supports fully automatic instrumentation of all usages of clocks, finishes, and the SPMDBarrier, including their distributed versions. All of the key information required by Armus, such as task IDs and the barrier membership of each task, is directly obtained from the X10 runtime. Our implementation uses a small extension that considers tasks as waiting on a set of events (i.e., a single task may have multiple wait-on relationships), to explicitly handle the X10 advanceAll command for synchronising on every clock that the calling task is registered with. Alternatively, we could treat advanceAll in Armus without these extensions, through its encoding into split-phase synchronisations (call resume on each clock in arbitrary order, followed by advance on each in arbitrary order).\footnote{http://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf §15.1.4}

**Distributed deadlock detection.** One of the key design goals of X10 is to promote a smooth migration between shared memory and distributed deployments of barrier programs [10]. A distributed barrier program is composed of tasks synchronising on shared barriers while running at different places, which may map concretely underneath to processes running in separate address spaces on the same, or different, machines, synchronising by asynchronous message passing. Listing 4 gives a distributed version of the code in Listing 1 (corrected to avoid the original
Listing 4. Distributed X10 version of Listing 1 (corrected to avoid the original deadlock), executing each child task at a different remote place p.

```x10
val c = Clock.make();
finish {
  for (p in Place.places())
    at (p) async {
      for (j in 1..J) {
        val l = a(i-1);
        val r = a(i+1);
        c.advance();
        a(i) = (l+r)/2;
        c.advance();
      }
    }
  c.drop();
}
```

deadlock) where each child task is executed in a separate place, as designated by the at clause of the async statement.

A distributed deployment of Armus-X10 features an instance of the Armus runtime at each distributed site, with reliable access to a (remote) central data store: our implementation uses a TCP connection from each site to a failure resilient Redis server. Each Armus instance periodically uploads its local blocking status to the data store, i.e., a disjoint portion of the global system state. Likewise, the deadlock checker periodically, and asynchronously, polls the data store for the current snapshot of the system state, on which it performs the deadlock detection. Consistency is ensured by the use of TCP for ordered and reliable delivery of the state update messages between each site and the central store. We do not assume any synchronisation between the blocking status messages from different sites.

The basis for this approach is rooted in the basic methodology outlined in Section 6.1. First, note that it is sound to conduct the verification on the partial snapshot of system state formed from the blocking statuses of any subset of blocked tasks (i.e., due to asynchronous delays of update messages). Then the key point regarding overall soundness (that is, the key difference between this distributed setting and the basic methodology) relates to the clearing of blocking statuses from the central store when tasks complete their await operations. The asynchrony of these messages means that the deadlock detection may be conducted on a model that includes "dead" edges, i.e., those built from a blocking status for which the task has completed its await in the actual system. However, it is inherently impossible for such an edge to be part of a cycle in the model if the relevant task is no longer blocked in the actual system (i.e., a false positive scenario), since phaser deadlock is a stable property. The stability of deadlocks and assumption of reliable network infrastructure also ensures a form of completeness for this setting, in that a deadlock is always eventually detected. The correctness of the verification is thus unaffected by any discrepancy arising due to asynchrony, between the analysis state according to the central store and the concrete state of the actual system.
c = new Phaser(1); // "clock" Phaser (Java registration of parent task via arg value)
f = new Phaser(1); // "finish" Phaser
JArmus.register(c); // Additional JArmus "registration" (calling thread ID recorded)
JArmus.register(f);
for (int i = 1; i <= I; i++) {
    c.register(); // Java registration (API internal counter increment)
f.register();
    new Thread() { // Define and spawn i-th task
        public void run() {
            JArmus.register(c); // Additional JArmus "registration"
            JArmus.register(f);
            for (int j = 1; j <= J; j++) {
                l = a[i-1];
r = a[i+1];
c.arriveAndAwaitAdvance();
a[i] = (l + r) / 2;
c.arriveAndAwaitAdvance();
}
            c.arriveAndDeregister();
f.arriveAndDeregister();
    }.start();
}
c.arriveAndDeregister();
f.arriveAndAwaitAdvance();

6.4 JArmus
JArmus supports the standard Phaser API, and other barrier programming facilities in the
java.util.concurrent package that it subsumes, such as CountDownLatch and CyclicBarrier. The Java Phaser is a limited version of the general concept of phasers that supports dynamic
membership and split-phase synchronisations, but does not permit asynchronous advancing of
local phases by individual members (and, consequently, cannot support awaiting arbitrary phases).
This limitation is related to a design choice of these APIs that, unlike the X10 runtime, do not
record barrier membership with respect to an explicit notion of task (i.e., thread) ID. Instead, a
Phaser simply records the number of times the register method is called, without considering
the identity of calling threads. It is left to the programmer to use register appropriately, and
hence the relationship between the threads that called register and the actual participants of a
synchronisation (i.e., threads calling one of the await methods) is also left implicit. Similarly for
CyclicBarrier, the programmer declares the number of participants (and shares the object with
that many tasks), but does not specify which tasks participate in synchronisations.

Due to the above limitation, JArmus, unlike Armus-X10, does not support fully automatic
instrumentation of Java programs that use these APIs. JArmus instead relies on the programmer to
manually provide the missing thread membership information by additionally calling the static
register method of the JArmus class, typically on task start up (cf. the clocked clause in an X10

http://redis.io/
async). For example, Listing 5 lists a Java version of the running example from Listing 1 using Phaser and JArmus. The Java “registration” of each child task to, e.g., the “clock” Phaser c (Line 6), is matched by a JArmus.register, taking the phaser as an explicit argument and the ID of the calling thread implicitly, at the start of the task (Line 10). The Java registration of the parent task, implicitly signified by initialising the phaser to a count of 1 (Line 1), is similarly matched by an explicit JArmus.register (Line 3).

In general, there is no precise method for statically inserting these JArmus calls automatically, nor any way to reconstruct the missing membership information at run-time from the existing Phaser (or CountDownLatch, CyclicBarrier) classes alone. In practice, if the user does not correctly use JArmus.register to register a task to some phaser, then a JArmus runtime exception will typically be raised if and when the task attempts a relevant operation on the phaser, due to the Armus instrumentation of the latter.

The information on which tasks are participating, rather than just counting the number of participants, would be a crucial requirement to extend Java Phaser to support all synchronisation patterns possible with phasers. For instance, in a multi-producer-single-consumer pattern, the phase number of each producer allows the consumer to proceed step-wise at the pace of whichever is the “slowest” producer, and which may vary throughout the computation.

7 EVALUATION

The aim of the evaluation process is to 1) ascertain whether the performance impact of Armus scales with the increase in the number of tasks, 2) evaluate the performance overhead of distributed deadlock detection, and 3) compare execution impact of selecting between the SG with the WFG and using the dynamic model selection approach.

The hardware used to run the benchmarks has four AMD Opteron 6376 processors, each with 16 cores, making a total of 64 cores. There are 64GB of available RAM. The operating system used is Ubuntu 13.10. For the languages, we used Java build 1.8.0_05-b13, and X10 version 2.4.3. For compiling and running we used the default compiler and runtime flags of each benchmark suite.

We follow the start-up performance methodology detailed in [26]. We take 31 samples of the execution time of each benchmark and discard the first sample. Next, we compute the mean of the 30 samples with a confidence interval of 95%, using the standard normal z-statistic.

7.1 Impact of Non-distributed Verification

The two goals of this evaluation are: to measure the impact of verification on standard Java benchmarks, and ii) to measure whether the verification scales with the increase of the number of tasks. We run the verification algorithm against a set of standard parallel benchmarks available for Java. JArmus is run in the detection mode (every 100 milliseconds) and in the avoidance mode, both use the dynamic model selection. Note that the Java applications we checked are not distributed.

We select benchmarks from the NAS Parallel Benchmark (NPB) suite [24] and the Java Grande Forum (JGF) [65] benchmark suite. The NPB ranges from kernels to pseudo-applications, taken primarily from representative Computational Fluid Dynamics (CFD) parallel applications. The JGF is divided into three groups of applications: micro-benchmarks, computational kernels, and pseudo-applications. All benchmarks proceed iteratively, and use a fixed number of cyclic barriers to synchronise stepwise. Furthermore, all benchmarks check the validity of the produced output.

For the sake of reproducibility we list the parameters of the benchmarks run as specified in [24, 65]: BT uses size A, CG uses size C, the Java version of FT uses size B, MG uses size C, RT uses B, and SP uses size W. Note: the input set chosen for benchmark SP only allows it to scale up to 31 tasks; however, to simplify the presentation of the graphs, we have represented the results of this benchmark in the 32-task category.
Figure 7 summarises the comparative study of the execution time for each benchmark. The results for the NPB and JGF benchmark suites are depicted in Figures 7a to 7f. In detection mode, since there is a dedicated task to perform verification, we observe that the overhead does not increase linearly as we add more tasks. The runtime-factor sits below $1.15 \times$ and in most cases is negligible. In avoidance mode, each task checks the graph whenever it blocks, so as we add more tasks, the execution overhead increases. Still, in the worst case, benchmark CG, the runtime-factor is $1.50 \times$, which is acceptable for application testing purposes.
7.2 Impact of Distributed Verification

The goal of the evaluation is to measure the runtime overhead of deadlock detection in available X10 distributed applications. Armus-X10 is configured with the distributed deadlock detection mode, running the verification algorithm every 200 milliseconds. The chosen benchmarks are available via the X10 source code repository. Deadlock avoidance is unavailable in the distributed setting.

Benchmarks FT and STREAM come from the HPC Challenge benchmark [46], SSAC2 is an HPCS Graph Analysis Benchmark [3], JACOBI and KMEANS are available from the X10’s website. For reproducibility purposes the non-default parameters we select are: FT magnitude 11; KMEANS 25k points, 3k clusters to find, and 5 iterations; JACOBI matrix of size 40, maximum iterations are 40; SSCA2 215 vertices, a with a probability of 7%, and no permutations; STREAM with size of 524k.

Figure 8 depicts the execution time of each benchmark with and without verification. There is no statistical evidence of an execution overhead with running deadlock detection mode.

7.3 Impact of the Graph Model Choice

The goal of this evaluation is to measure the impact of the graph model in the verification procedure. To this end we analyse the worst case behaviour: programs that generate graphs with thousands of edges. In particular, we evaluate our dynamic model selection against the usual static model selection (WFG and SG).

We select a suite of programs that spawn tasks and create barriers as needed, depending on the size of the program, unlike the classical parallel applications we benchmark in Sections 7.1 and 7.2 where the number of tasks should correspond to the number of available processing units (cores). The suite of programs exercises different worst case scenarios for the verification algorithm: many tasks versus many barriers.

The chosen benchmarks are educative programs taken from the course on Principles and Practice of Parallel Programming, taught by Martha A. Kim and Vijay A. Saraswat, Fall 2013. BFS performs a parallel breadth-first search on a randomly generated graph. There is a task per node being visited and a barrier per depth-level of the graph. FI computes a Fibonacci number iteratively with a shared array of clocked variables (each pairs a barrier with a number). Each element of the array holds the outcome of a Fibonacci number. When the program starts it launches n tasks. The i-th

---

18 http://sourceforge.net/projects/x10/files/x10/2.4.3/x10-benchmarks-2.4.3.tar.bz2/download
19 http://www.cs.columbia.edu/~martha/courses/4130/au13/
Table 3. Average edge count per benchmark per graph mode.

<table>
<thead>
<tr>
<th></th>
<th>SE</th>
<th>FI</th>
<th>FR</th>
<th>BFS</th>
<th>PS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto</td>
<td>24</td>
<td>808</td>
<td>190</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>SG</td>
<td>53</td>
<td>2143</td>
<td>1735</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>WFG</td>
<td>24</td>
<td>1285</td>
<td>89</td>
<td>605</td>
<td>789</td>
</tr>
<tr>
<td>TEG</td>
<td>74</td>
<td>2077</td>
<td>1643</td>
<td>1776</td>
<td>1450</td>
</tr>
</tbody>
</table>

Task stores its Fibonacci number in the $i$-th clocked variable and synchronises with task $i + 1$ and task $i + 2$ that read the produced value. FR computes a Fibonacci number recursively. Recursive calls are executed in parallel and a clocked variable synchronises the caller with the callee. SE implements the Sieve of Eratosthenes using clocked variables. There is a task per prime number and one clocked variable per task. PS computes the prefix sum—or cumulative sum—for a given number of tasks. Given an input array with as many elements as there are tasks, the outcome of task $i$ is the partial sum of the array up to the $i$-th element. All tasks proceed stepwise and are synchronised by a global barrier.

Figures 9 and 10 depict the execution time of each benchmark verified by Armus-X10 in avoidance and detection modes (respectively) where we vary the selection method of the graph model. Table 3 lists the average number of edges used in verification and the relative execution time overhead of each benchmark. When running in detection, since there is no statistical difference in the average overhead (cf. Figure 10), Table 3 simply lists the verification overhead of auto mode when running in avoidance mode only.

We can classify the benchmarks in three groups according to the ratio between the number of tasks and the number of resources: i) similar count of tasks and resources, benchmark SE; ii) much more resources than tasks, benchmarks FI and FT; and iii) much more tasks than resources, benchmarks BFS and PS. When i) there are as many resources as there are tasks, then all graph models perform equally well. When ii) there are more resources than tasks, and iii) vice-versa, the choice of the graph model is of major importance for a verification with low impact on the execution time.

Overall, dynamic graph selection outperforms static graph selection. Furthermore, the worst model to choose from is the bipartite TEG graph, as it contains more information than the WFG and the SG. When considering dynamic graph selection, the worst-case runtime-factor for deadlock detection is $1.2 \times$ and $2.4 \times$ for deadlock avoidance. The graph model choice severely amplifies the verification overhead in deadlock avoidance. The case in point is benchmark PS; the runtime-factor for dynamic selection is $1.8 \times$, for SG is $2.6 \times$, for WFG is $5.9 \times$, and for TEG is $14.4 \times$.

8 RELATED WORK

This section lists related work focusing on deadlock verification in parallel programming languages. Background on graph-based approaches to deadlock detection were discussed in Section 4.1.

Deadlock prevention. The literature around source code analysis to prevent barrier related deadlocks is vast. The fork/join programming model is easily restricted syntactically to prevent deadlocks from happening. Lee and Palsberg [43] present a calculus for a fork/join programming model, suited for inter-procedural analysis through type inference, and establish a deadlock freedom property. The work also includes a type system that is used to identify may-happen-parallelism, further
explored in [1]. Finally, related work on “barrier matching” tackles the problem of barrier deadlocks in a setting where there is only global barrier synchronisation [39, 73].

Cogumbreiro et al. [14] propose a static typing system to ensure the correctness of phased activities for a fragment of X10 that disallows awaiting on a particular clock. Therefore, programs that involve more than one clock and that perform single waits cannot be expressed, nor verified (cf. the X10 and Java programs we present in Section 2).

The tool X10X [28] is a model checker for X10. Model checkers perform source code analysis and can be used to discover potential deadlocks. This class of tools is affected by the state explosion problem: the analysis grows exponentially with the number of possible interleaves of the program. Thus, X10X may not be able to verify complex programs. In general, prevention is too limiting to be applied to the whole system, so language designers use this strategy to eliminate just a class of deadlocks.

Ganjiei et al. propose a static verification technique for unbounded phaser synchronisation [25]. The proposed tool performs symbolic execution on a simple language with branching and conditional loops. The authors show that the problem of static deadlock freedom for such a language is undecidable.

**Deadlock avoidance.** The problem of deadlock avoidance is a very well studied problem that dates as far back as 1960’s, e.g., Banker’s Algorithm by Edsder Dijkstra [22]. For instance, Minoura [47] and Reveliotis et al. [56] cover the problem complexity in deadlock avoidance with intricate synchronisation patterns. In general, deadlock avoidance can only disallow actions that lead to a deadlock and inform the culprit task of its error, e.g., Armus throws an unchecked exception. For some synchronisation mechanisms, however, it is possible to preclude schedules that may lead to a deadlock by: controlling the lock acquisition order [6, 27, 51, 70]; using transactions to avoid data races which lead to deadlocks with futures [50, 72]; executing critical regions as transactions [55]; and adding extra data in streaming computation [45]. To our best knowledge, techniques that avoid deadlocks in the context of barrier synchronisation only handle a few situations of barrier deadlocks, unlike our proposal that is complete (with reference to Theorem 5.11). For instance, in X10 and HJ, tasks deregister from all barriers upon termination; this mitigates deadlocks that arise from missing participants. HJ avoids deadlocks that originate from the interaction between phasers.
and finish blocks by limiting the use of phasers to the scope of finish blocks. Cogumbreiro et al. use Armus in the context of a tool that specialises in avoiding deadlocks caused by futures [16].

**Deadlock detection.** UPC-CHECK [57] deals with deadlock detection, but in a simpler setting where barriers are global; in contrast, our work can handle group synchronisation. Literature concerning MPI deadlock detection takes a top-down approach: the general idea is given, but mapping it to the actual MPI semantics is left out. DAMPI [69] reports a program as deadlocked after a period of inactivity, so it may indicate false positives, i.e., it can misidentify a slow program as being deadlocked. Umpire [32] and MUST [34] (a successor of Umpire) use a graph-based deadlock detection algorithm that subsumes deadlock detection to cycle detection, but omit a formal description on how the graph is actually generated from the language, cf. Theorems 5.9 and 5.11. We summarise the distributed detection technique of MUST. First, all sites collaborate to generate a single stream of events to a central site. The difficulty lays in ordering and aggregating the events generated by the various tasks. Then, the central site processes the stream of events to perform the collective checking, where, among other things, it identifies any completed barrier synchronisations. Finally, since MUST maintains a distributed wait state, the site performing the collective checking must broadcast the status of terminated synchronisations back to the various sites of the application. The wait state is required to delay the graph analysis as much as possible. In our approach, tasks only requires local information to maintain data consistency, which means that, in a distributed setting, Armus does not require the last synchronisation step that MUST performs. Furthermore, unlike MUST, Armus is capable of verifying split-phase synchronisation, known in MPI as non-blocking collective operations.

**Transitive closure.** Instead of testing whether the wait-for dependencies are cyclic (such as Armus does), one can test if a given blocked task can reach itself through the wait-for dependencies. The reachability problem can be solved by maintaining the transitive closure of the reachability relation on the wait-for graph. Such a technique has been used in the context of deadlock avoidance [5], yet the theoretical bounds are worst when compared to cycle detection. Computing the transitive closure from scratch can be solved with matrix multiplication [48]; the best known algorithm solves this problem in $O(n^{2.376})$ [17]. Alternatively, the transitive closure can be maintained dynamically [20], but updating the graph takes $O(n^3)$ time. Furthermore, maintaining the transitive closure usually assumes a fixed set of vertices throughout the execution, and the problem is compounded since updates and tests run concurrently.

**Verification of other barrier properties.** Saraswat and Jagadeesan [59] formalise the concurrency primitives of X10. Le et al. [42] devise a verification for the correct use of a cyclic barrier in a fork/join programming language. Vasudevan et al. [68] perform static analysis to improve performance of synchronisation mechanisms. Cogumbreiro et al. [15] formalises the Habanero phasers and a causality relation on phasers; the results are mechanised using the Coq proof assistant. Crafa et al. [18] present a small-step semantics of X10 with support for fault tolerance; the formalisation omits clocks (which are similar to phasers). The results of the paper are mechanised using the Coq proof assistant. Murthy et al. [49] propose the design of a distributed phaser, using skip lists. Scenarios of the distributed protocol are verified with the SPIN model checker.

9 CONCLUSION

We put forward Armus, a dynamic verification tool for barrier deadlocks that features both detection and avoidance, distribution support, and scalability improvements based on dynamic graph model selection. The target of verification is the core language BRENNER, introduced to represent programs with various barrier synchronisation patterns. The graph-based deadlock verification of Armus is
formalised and shown to be sound and complete against BRENNER. We prove that one can select from any of two graph models (WFG and SG) and correctly identify a deadlock situation. This result lets our tool dynamically choose the model that yields a smaller graph — a novelty in checking for deadlocks. Our benchmarks show that dynamic model selection outperforms the standard static model selection. Overall, the worst-case runtime-factor for deadlock detection is $1.21 \times$, and is often not statistically significant, e.g., in distributed benchmarks. We present two applications: Armus-X10 monitors any unchanged X10 program for deadlocks; JArmus is a library to verify Java programs. To the best of our knowledge, our work is the first dynamic verification tool that can correctly detect Java and X10 barrier deadlocks.

For future work, our goal is to extend the verification of our implementation. Our starting point is to verify the algorithm for distributed deadlock detection. Another direction is the verification of MPI programs that introduce complex patterns of point-to-point synchronisation and enable a direct comparison with state-of-the-art in barrier deadlock detection.

ACKNOWLEDGMENTS

We thank Olivier Tardieu and the PPoPP reviewers for their comments and suggestions on an earlier version of this work. The work is partially supported by EPSRC EP/K034413/1, EP/K011715/1, EP/L00058X/1, EP/N027833/1, and EP/N028201/1; and by EU FP7 612985 (UPSCALE). Tiago Cogumbreiro’s work on this paper was supported in part by the National Science Foundation under Collaborative Grant No. 1302570. Francisco Martins’ work on this paper was partially supported by FCT funding through LASIGE Research Unit, ref. UID/CEC/00408/2013.

REFERENCES


[56] Rahul Sharma, Michael Bauer, and Alex Aiken. 2015. Verification of Producer-consumer Synchronization in GPU Programs. In *PLDI*. ACM, 88–98. DOI: http://dx.doi.org/10.1145/2737924.2737962


ACM Transactions on Programming Languages and Systems, Vol. 1, No. 1, Article 1. Publication date: January 2018.


