A version of this technical report has been accepted for publication at ATVA 2013

An Expressive Framework for Verifying Deadlock Freedom

Duy-Khanh Le, Wei-Ngan Chin, Yong-Meng Teo

Department of Computer Science, National University of Singapore
{leduykha,chinwn,teoym}@comp.nus.edu.sg
(Technical Report - June 2013)

Abstract. This paper presents an expressive specification and verification framework for ensuring deadlock freedom of shared-memory concurrent programs that manipulate locks. We introduce a novel *delayed lockset checking* technique to guarantee deadlock freedom of programs with interactions between thread and lock operations. With disjunctive formulae, we highlight how an abstraction based on *precise lockset* can be supported in our framework. By combining our technique with locklevels, we form a unified formalism for ensuring deadlock freedom from (1) double lock acquisition, (2) interactions between thread and lock operations, and (3) unordered locking. The proposed framework is general, and can be integrated with existing specification logics such as separation logic. Specifically, we have implemented this framework into a prototype tool, called PARAHIP, to automatically verify deadlock freedom and correctness of concurrent programs against user-supplied specifications.

Keywords: Concurrency, Deadlock, Specification, Verification.

1 Introduction

Concurrent software systems are often complex, error-prone, and require tremendous efforts from programmers to make them work correctly [24]. Over the past decade, verification has been viewed as one of the solutions to this challenging research problem on increasing the quality and reliability of (concurrent) programs, as advocated by Tony Hoare [13]. However, understanding and reasoning about the correctness of concurrent programs is rather complicated due to non-deterministic interleavings of concurrent threads [21]. These interleavings may result in *deadlocks* [6], i.e. states in which each thread in a set blocks waiting for another thread in the set to release a lock or complete its execution. Deadlocks are common defects in software systems. Specifically, in Sun's bug report database at `http://bugs.sun.com/`, there are approximately 6,500 bug reports out of 198,000 ($\sim$ 3%) containing the keyword "deadlock" [27]. In this paper, we propose an expressive framework for reasoning about the correctness of concurrent programs with a focus on eliminating deadlocks.

Existing verification systems [11, 14, 22, 23] often use *abstract predicates* to represent states of locks. For example, Gotsman et al. [11] use abstract predicate *Locked(x)* to specify that the lock x is owned by the current thread. Hobor et al. [14] use the predicate *hold x R* and CHALICE [22, 23] uses *holds(x)* for the same purpose. Intuitively, a lock is owned by a thread if it is in the set of locks

already acquired by the thread, i.e. the thread's lockset. Interestingly, although using predicates, previous works [11, 14, 22, 23] formulate their soundness proof using the notion of lockset. Additionally, Haack et al. [12] show that lockset (or rather lockbag) is necessary to reason about Java recursive locks. In retrospect, one can say that lockset has proven to be an important abstraction for verifying concurrent programs that manipulate locks.[1] In this paper, we advocate the use of *precise locksets* for explicitly reasoning about the presence or absence of locks, empowering a more expressive framework for verifying deadlock freedom even in the presence of interactions between thread operations (e.g. fork/join) and lock operations (e.g. acquire/release). Due to the non-deterministic nature of threads, sound reasoning of the interactions between thread and lock operations is non-trivial.

```
1  int running;
2  pthread_t thread;
3  mutex_t mutex;
4
5  void* timer(){
6    int state;
7    do{
8      mutex_lock(&mutex);
9      state=running;
10     mutex_unlock(&mutex);
11     .../*timing*/
12   }while(state);
13 }
14 void main(){
15   running = 0;/*init timer*/
16   mutex_lock(&mutex);
17   running = 1;/*start timer*/
18   pthread_create(&thread,&timer);
19   mutex_unlock(&mutex);
20   /*begin timed computation*/
21   ...
22   /*end computation*/
23   mutex_lock(&mutex);
24   running = 0;/*stop timer*/
25   mutex_unlock(&mutex);
26   pthread_join(thread);
27 }
```

Fig. 1. A program with interactions between thread and lock operations

Fig. 1 outlines a simplified[2] C implementation of a timer used in NetBSD operating system's report database [1]. Though rather intricate due to the interactions between lock and thread operations, the program is deadlock-free because the two threads never wait for each other. However, if the programmer does not release the lock before joining (e.g. line 25 is missing or line 25 and 26 are swapped), the interactions will cause a deadlock when the main thread blocks waiting to join the child thread and the child thread also blocks waiting to acquire the mutex being held by the main thread. For larger programs with many (possibly non-deterministic) execution branches, these interactions are not easy to follow [21]. With concurrent programs becoming mainstream in this multicore era, we will increasingly require a more comprehensive solution for constructing and verifying these intricate interactions.

In this paper, we propose an expressive verification framework to guarantee deadlock freedom in the presence of such interactions. Our framework has the following innovations:

[1] See Appendix A for detailed comparison between abstract predicates and locksets.

[2] In the original implementation, there is a conditional variable associated with the mutex to more efficiently signal the `timer` thread to start and stop timing. As verifying conditional variables is an orthogonal issue, we have omitted them for simplicity.

– *Delayed lockset checking* to help reason about the interactions between thread and lock operations. Unlike the traditional verification approaches [11, 12, 14, 15, 22] that check pre-conditions of procedures entirely at fork points, this technique allows lockset constraints in the pre-conditions to be delayed and checked at join points instead. This prevents deadlocks due to the interactions and also permits more programs to be declared as deadlock-free.
– *Precise lockset reasoning*, as opposed to ones based on *abstract predicates* or *approximated locksets*, to ensure that deadlock-free pre-conditions on lock acquisition and release can be guaranteed. Any uncertainty, if any, from static program analysis is simply captured through the use of explicit disjunction.
– *Combining lockset with the concept of locklevels* in the literature [3, 22, 33] to form an expressive framework for ensuring deadlock freedom, covering various scenarios such as double lock acquisition, interactions between thread and lock operations, and unordered locking.
– A prototype specification and verification system, called ParaHIP, to show that the proposed framework has been successfully integrated with separation logic [31] for reasoning about concurrent programs.

The rest of this paper is organized as follows. Section 2 gives concrete examples that motivate our delayed lockset checking technique and show how precise lockset reasoning can be systematically supported. Section 3 presents our specification logic for verification. Section 4 shows our verification rules and presents our soundness guarantee on deadlock freedom. Section 5 discusses the implementation and experimental results of our prototype tool. Section 6 summarizes related work. Section 7 concludes our paper.

2 Motivation and Proposed Approach

2.1 Precise Lockset Reasoning

As our proposal is language-independent, we have developed a core language (described in Section 3) to capture the basic ideas. In the rest of the paper, we shall express our examples using this core language. In our verification framework, **LS** is a thread-local ghost variable[3] capturing the set of locks held by a thread. Lockset is a verification concept rather than a programming language concept. Using lockset, verification rules for *acquire* and *release* operations on non-recursive (mutex) locks[4] can be defined as follows:

```
acquire(lock x)
  requires x∉LS
  ensures LS′=LS∪{x};
```

```
release(lock x)
  requires x∈LS
  ensures LS′=LS−{x};
```

Note that we use *primed notation* to denote updates to variables. The primed version $\mathbf{LS}'$ of the variable **LS** denotes its latest value; the unprimed version **LS** denotes its old value at the start of the respective procedure call. Using lockset,

[3] Ghost variables are variables used for verification purpose. They do not affect program correctness.
[4] Cannot be acquired more than once; also called non-reentrant locks

```
void thread()
  requires LS=={}  ensures LS'=={};
{
  lock l1 = new lock();
  //{ LS'=={} }
  acquire(l1);
  //{ LS'=={l1} }
  func(l1); /*Error*/
  release(l1);
}
```

```
void func(lock l1)
  requires l1∉LS  ensures LS'=LS;
{
  //{ l1∉LS ∧ LS'=LS }
  acquire(l1);
  //{ l1∉LS ∧ LS'=LS∪{l1} }
  release(l1);
  //{ l1∉LS ∧ LS'=LS∪{l1}−{l1} }
  //{ l1∉LS ∧ LS'=LS }
}
```

Fig. 2. Deadlock due to double acquisition of a non-recursive lock

it is straightforward to prevent the deadlock due to acquiring a non-recursive lock twice in the `thread` code of Fig. 2. In this sequential setting, our verification reports an error because the pre-condition of the callee `func` ($l1 \notin \mathbf{LS}$) cannot be satisfied by the current lockset of the caller ($\mathbf{LS}'=\{l1\}$). Additionally, the `release` rule excludes the possibility of releasing a lock more than once.

In each given program, there can be many locking scenarios across different execution branches. Each branch could potentially have a different lockset. The following code fragment shows a simple example where locksets at two branches are $\mathbf{LS}'=\{x\}$ and $\mathbf{LS}'=\{\}$, which are clearly different:

```
//{ LS'={} }
if (b) { acquire(x);//{ LS'={x} } } else { //{ LS'={} } }
```

For static analysis, we often perform some approximation. For example, one may *over-approximate* on the lockset, by using $\mathbf{LS}'=\{x\}$ as the post-state of the above code fragment. However, this approach would fail to detect the definite presence of the lock `x` for safe release. Another approach is to *under-approximate* on the lockset by using $\mathbf{LS}'=\{\}$, but this approach fails to detect the definite absence of the lock for safe acquisition. Thus, one plausible solution is to combine the two approximations by capturing both may-hold and must-hold locksets, simultaneously. However, this approach would be more complex due to the use of two locksets. In this paper, we propose a simpler solution that would mandate the use of *precise locksets* in our verification/analysis. For approximation, we propose to use *disjunctive formulae* to capture uncertainty and also allow program states, other than lockset, to be over-approximated. In the above example, we can ensure precise lockset by using either $b \wedge \mathbf{LS}'=\{x\} \vee \neg b \wedge \mathbf{LS}'=\{\}$ or even $\mathbf{LS}'=\{x\} \vee \mathbf{LS}'=\{\}$ as its post-state, but never $\mathbf{LS}'=\{x\}$, since we always ensure that each lockset is precisely captured and never approximated. This principle allows us to support precise reasoning on locksets for verifying deadlock freedom.

2.2 Delayed Lockset Checking

Fig. 3 shows two programs that pose challenges for existing verification systems. The programs are challenging because they express rich interactions between fork/join concurrency and lock operations. The traditional way of verification [11, 12, 14, 15, 22] cannot sufficiently handle these scenarios because it performs the check for the pre-condition of the forkee *only* at the *fork point*. This could

```
void func(lock l1)
  requires l1∉LS  ensures LS′=LS;
{ acquire(l1); release(l1); }

void main()
  requires LS={}  ensures LS′={};
{
 lock l1 = new lock();
 //{ LS′={} }
 int id = fork(func,l1);/*DELAY*/
 //{ LS′={} }
 acquire(l1);
 //{ LS′={l1} }
 /*Potentially deadlocked when join*/
 join(id); /*CHECK, error*/
 release(l1);
}
```

(a) Potentially deadlocked

```
void func(lock l1)
  requires l1∉LS  ensures LS′=LS;
{ acquire(l1); release(l1); }

void main()
  requires LS={}  ensures LS′={};
{lock l1 = new lock();
 //{ LS′={} }
 acquire(l1);
 //{ LS′={l1} }
 int id = fork(func,l1);/*DELAY*/
 //{ LS′={l1} }
 release(l1);
 //{ LS′={} }
 join(id);/*CHECK, ok*/
 //{ LS′={} }
}
```

(b) Deadlock-free

Fig. 3. Examples of programs exposing interactions between thread and lock operations

incorrectly verify the program in Fig. 3a as deadlock-free and reject the deadlock-free program in Fig. 3b. The well-known technique [3, 22, 33] which requires threads to acquire multiple locks in a specific order to avoid deadlocks could not directly handle complications due to fork/join concurrency. In this paper, we propose *delayed lockset checking* technique that is capable of preventing deadlock scenarios (such as that presented in Fig. 3a) and proving more programs (such as that described in Fig. 3b) to be deadlock-free.

This technique is based on the following observation. At a *fork point*, a verifier is unaware of future operations performed by a main (or parent) thread; the only information it knows of is future locking operations executed by a child thread thanks to the use of *lockset*. For example, a constraint $l1 \notin \mathbf{LS}$ in the pre-condition of a child thread implies that the child thread is going to acquire the lock `l1`. Therefore, in order to ensure that the child thread will finally be able to acquire the lock (and thus avoid deadlocks), the main thread should not be holding the lock while waiting for the child thread at its *join point*. In other words, *when forking a child thread, lockset constraints in its pre-condition are not checked at the fork point but are delayed to be checked at its join point instead.*

The deadlock in Fig. 3a can be prevented by deferring the lockset constraint $l1 \notin \mathbf{LS}$ of the child thread to its join point. At the join point, the constraint is checked and the verification reports an error because the constraint is unsatisfiable ($\mathbf{LS}'=\{l1\}$ at the join point). Similarly, the program in Fig. 3b is ensured as being deadlock free because the lockset constraint $l1 \notin \mathbf{LS}$ is delayed from the fork point and is satisfiable at the join point ($\mathbf{LS}'=\{\}$). Note that, although main and child threads have different locksets, a constraint $l1 \notin \mathbf{LS}$ in pre-conditions of a child thread indicates its intention to acquire the lock $l1$, hence this constraint can be soundly checked against the lockset of the main thread to prevent deadlocks. Besides, it is unsound to check lockset constraints at any satisfiable

points in the middle of the fork point and the join point. For example, in a scenario similar to Fig. 3b, after forking a child thread, the main thread releases the lock. At this point, the lockset constraint is satisfiable. However, the main thread could later acquire the lock again and wait for the child thread to join. This scenario still suffers a potential deadlock. As a result, it is only sound to check delayed lockset constraints at just the join points.

In summary, the main benefit of our *delayed lockset checking* technique is to facilitate more expressive deadlock verification in the presence of interactions between parent/child threads and lock operations.

2.3 Combining Lockset and Locklevel

Another type of deadlocks occurs when threads attempt to acquire the same set of locks in different orders (unordered locking). An example of such a scenario is shown in Fig. 4. Locklevel is a well-known handle to prevent deadlocks due to unordered locking [3, 22, 33]. Intuitively, each lock in a program is associated with a ghost field *mu* representing the lock's level. For example, $l1.mu$ denotes the locklevel of lock `l1`. With it, deadlocks can be prevented indirectly by ensuring that locks are acquired in a strictly increasing order of locklevels. To check that locks are acquired in the specified order, a ghost variable **waitlevel** is used to capture the maximum level currently acquired by a thread, i.e. **waitlevel** is the maximum level among locklevels of all locks in current thread's lockset **LS**. A thread can acquire a lock only if its current waitlevel **waitlevel**$'$ is lower than the lock's level. Using locklevels, the deadlock in Fig. 4 can be prevented. The verification system reports an error when the child thread attempts to acquire lock `l1` whose locklevel is lower than the current waitlevel of the child thread.

In the pre-condition of the `func` procedure (Fig. 4), we use the specification $[\omega\#\psi]$ to capture the fact that the waitlevel constraint ω and the lockset constraint ψ are *mutually exclusive*, i.e. the former is checked in sequential settings, while the latter is a check needed to be delayed in concurrent settings. This

```
void main()
 requires LS={}  ensures ...;
{lock l1,l2 = new lock();
 assume(l1.mu<l2.mu);
 // { LS'={}∧l1.mu<l2.mu
 //   ∧waitlevel'=0 }
 int id = fork(func,l1,l2);
 // { LS'={}∧l1.mu<l2.mu
 //   ∧waitlevel'=0 }
 acquire(l1);
 // { LS'={l1}∧l1.mu<l2.mu
 //   ∧waitlevel'=l1.mu }
 acquire(l2);
 // { LS'={l1,l2}∧l1.mu<l2.mu
 //   ∧waitlevel'=l2.mu }
 ...}
```

```
void func(lock l1,lock l2)
 requires [waitlevel<l1.mu # l1∉LS∧l2∉LS]
          ∧ l1.mu<l2.mu
 ensures ...;
{
 // { waitlevel'<l1.mu ∧ l1.mu<l2.mu
 //   ∧ LS'=LS }
 acquire(l2);
 // { waitlevel'=l2.mu ∧ l1.mu<l2.mu
 //   ∧ LS'=LS ∪ {l2} }
 acquire(l1); /*Error*/
 ...
}
```

Fig. 4. A potential deadlock due to unordered locking

provides a *single* mechanism for procedure declarations so that each procedure could be either forked as a child thread or invoked as a normal procedure call.

In summary, precise lockset, delayed lockset checking, and locklevel are complementary and combining them is essential to form an expressive framework for verifying various deadlock scenarios such as double acquisition, interactions between fork/join and acquire/release, and unordered locking.

3 A Specification Logic for Deadlock Freedom

In this section, we present a specification logic that can be used to verify deadlock freedom. We show how our approach, based on precise lockset abstraction, can be integrated with the locklevel idea from Chalice [22]. We also present a specification formalism to unify constraints on lockset, locklevel and waitlevel into a single specification and to allow each procedure to be used internally or as the entry point of a newly-forked thread.

P	::= $proc^*$	Program
$proc$	::= $\texttt{pn}((\texttt{[ref]}\ t\ v)^*)\ \ spec^*\ \{\ s\ \}$	Procedure declaration
$spec$	::= **requires** Φ_{pr} **ensures** Φ_{po};	Pre/Post-conditions
t	::= `int` | `bool` | `void` | `lock`	Type
e	::= $v \mid k \mid e_1{=}e_2 \mid e_1{\neq}e_2 \mid \ldots$	Expression
	$v = \textbf{fork}(pn,v^*) \mid \textbf{join}(v)$	
	$\mid \texttt{lock}\ v = \textbf{new}\ \texttt{lock}(v)$	
s	::= $\mid \textbf{acquire}(v) \mid \textbf{release}(v)$	Statement
	$\mid s_1; s_2 \mid \texttt{pn}(v^*) \mid \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2$	
	$\mid \ldots$	

Fig. 5. Programming Language with Annotations and Concurrency

Programming Language We consider an imperative core language (Fig. 5) with fork/join concurrency for dynamic thread creation and non-recursive locks. The language is relative straightforward; its details are described in Appendix B.

3.1 Integrating Specification with LockLevels

In our specification logic, a lockset variable **LS** captures a set of locks held by the current thread. Like Chalice [22], each lock in a program has an immutable ghost field *mu* representing the lock's level. Locklevels are implemented as natural numbers and operators $=$, $<$ and $>$ are used over locklevels. The lowest (bottom) locklevel is denoted as 0. A **waitlevel** variable can be derived from the lockset and locklevels. As a reminder, waitlevel is the maximum level among locklevels of all locks in current thread's lockset **LS**. Levels of locks in a program are strictly positive while a bottom locklevel denotes the waitlevel in case of empty lockset. Using lockset as an abstraction, constraints on waitlevel can be expressed in terms of constraints on lockset and locklevels as follows:

$$\textbf{waitlevel}{<}x \stackrel{\texttt{def}}{=} (\textbf{LS}{=}\{\} \Rightarrow 0{<}x) \wedge (\textbf{LS}{\neq}\{\} \Rightarrow \forall v{\in}\textbf{LS} \cdot v.mu{<}x)$$
$$\textbf{waitlevel}{>}x \stackrel{\texttt{def}}{=} (\textbf{LS}{=}\{\} \Rightarrow 0{>}x) \wedge (\textbf{LS}{\neq}\{\} \Rightarrow \exists v{\in}\textbf{LS} \cdot v.mu{>}x)$$
$$\textbf{waitlevel}{=}x \stackrel{\texttt{def}}{=} (\textbf{LS}{=}\{\} \Rightarrow 0{=}x) \wedge (\textbf{LS}{\neq}\{\} \Rightarrow \forall v{\in}\textbf{LS} \cdot v.mu{\leq}x \wedge \exists u{\in}\textbf{LS} \cdot u.mu{=}x)$$

$$
\begin{array}{rrl}
\text{Logic formula} & \Phi & ::= \bigvee(\exists v^* \cdot \mu[(\textbf{and}\ \tau)^*]) \\
\text{Main thread formula} & \mu & ::= \ell \wedge \pi \\
\text{Child thread formula} & \tau & ::= \textbf{thread}{=}v \wedge \gamma \rightsquigarrow_{\{w^*\}} \pi \\
\text{Lock formula} & \ell & ::= [\bigwedge\omega \ \#\ \bigwedge\psi] \\
\text{Delayed formula} & \gamma & ::= \bigvee(\bigwedge\psi \wedge \pi) \\
\text{Waitlevel formula} & \omega & ::= \textbf{waitlevel}{=}\alpha^t \mid \textbf{waitlevel}{<}\alpha^t \mid \textbf{waitlevel}{>}\alpha^t \\
\text{Lockset formula} & \psi & ::= v \in \textbf{LS} \mid v \notin \textbf{LS} \\
\text{Pure formula} & \pi & ::= \alpha \mid \beta \mid \pi_1 \wedge \pi_2 \mid \pi_1 \vee \pi_2 \mid \neg\pi \mid \exists v \cdot \pi \mid \forall v \cdot \pi \mid \textbf{true} \\
\text{Set term} & \beta^t & ::= \textbf{LS} \mid \{\} \mid \{v\} \mid \beta_1^t \cup \beta_2^t \mid \beta_1^t \cap \beta_2^t \mid \beta_1^t - \beta_2^t \\
\text{Set formula} & \beta & ::= \beta_1^t \sqsubset \beta_2^t \mid \beta_1^t = \beta_2^t \\
\text{Arithmetic term} & \alpha^t & ::= k \mid v \mid v.mu \mid k \times \alpha^t \mid \alpha_1^t + \alpha_2^t \mid -\alpha^t \\
\text{Arithmetic formula} & \alpha & ::= \alpha_1^t = \alpha_2^t \mid \alpha_1^t \neq \alpha_2^t \mid \alpha_1^t < \alpha_2^t \mid \alpha_1^t \leq \alpha_2^t \\
& v, w \in \text{Variables} & \qquad k \in \text{Integer constants}
\end{array}
$$

Fig. 6. Grammar for Specification Language

3.2 Specification Formalism

Fig. 6 shows our specification logic. In the specification, Φ is a logic formula in disjunctive normal form. Each disjunct in Φ consists of a thread formula μ for a main (or parent) thread and a list of thread formulae τ (separated by the **and** keyword) to represent child threads. Each disjunct expresses the state of the main executing thread μ at a program point and the final states of its child threads τ. A main thread formula μ consists of a lock formula ℓ and a pure formula π. A lock formula ℓ consists of waitlevel formulae ω, and lockset formulae ψ. ω and ψ are self-explanatory.

A lock formula $[\bigwedge\omega \ \#\ \bigwedge\psi]$ presents our mechanism for each procedure's dual use, namely for both sequential and concurrent execution. The formula captures both waitlevel formula $\bigwedge\omega$ and lockset formula $\bigwedge\psi$ that are *mutually exclusive*. The former is checked for sequential procedural calls, while the latter must be delayed and checked at join points of forked threads. We provide both specifications in a unified format to cater to the differences in semantics for both sequential and concurrent computations. In sequential settings, e.g. when invoking a normal procedure call, the pre-condition of a procedure is an assertion that has to be fulfilled by the caller. If one or more constraints about lockset and waitlevel in the pre-condition are not met, verification fails. In concurrent settings and due to the ownership semantics of locks (see §10.1.2 of [4]), each new child thread does not inherit any locks from its parent thread. Hence, it has empty lockset and bottom waitlevel. Thus, constraints on waitlevel need not be checked here. Nevertheless, the constraints on lockset indicate the intention of the child thread and must be "delayed for checking" at its join point instead.

A child thread formula τ represents the final state of a child thread. It consists of a constraint $\textbf{thread} = v$ capturing the thread's identifier v, a delayed formula γ, and a pure formula π capturing the thread's post-state (i.e. its effects after finishing its execution). The formula τ denotes the fact that when a child thread with identifier v is joined and its delayed formula γ is satisfied, then its effects π will be visible to the calling thread. The annotation $\rightsquigarrow_{\{w^*\}}$ also captures a list of variables w^* that must be passed to the child thread when it is forked.

The formula γ illustrates our support for *delayed lockset checking*. Each disjunct in γ consists of delayed lockset constraints $\bigwedge\psi$ and a pure formula π to more precisely capture additional constraints for the corresponding delayed lockset constraints to hold. At each join point, only disjuncts whose pure formula is satisfied are candidates for delayed lockset checking.

Lastly, a pure formula π consists of standard equality/inequality, Presburger arithmetic and set constraints. Additionally, it is straightforward to enhance our specification logic with permissions to ensure data-race freedom by using separation logic [31] and variables as resource [30]. However, for simplicity of presentation, this paper shall focus on just the framework for deadlock freedom and ignore all issues pertaining to data-races.

For illustration, consider the following logic formula:

$$\exists v1, v2, tid \cdot\ l1{\neq}\texttt{null} \wedge l1.mu{=}v1 \wedge v1{>}0 \wedge l2{\neq}\texttt{null} \wedge l2.mu{=}v2 \wedge v2{>}0 \wedge id{=}tid \wedge \mathbf{LS'}{=}\{l2\} \wedge b \ \mathbf{and}\ \mathbf{thread}{=}tid \wedge ((l1{\notin}\mathbf{LS'} \wedge b \wedge l1{\neq}\texttt{null}) \vee (l2{\notin}\mathbf{LS'} \wedge \neg b \wedge l2{\neq}\texttt{null})) \rightsquigarrow_{\{l1,l2,b\}} \mathbf{true}$$

The formula represents a program state where there are two concurrent threads: a main thread currently holding the lock $l2$ and a child thread with identifier tid. The child thread has a disjunctive delayed formula which precisely captures two locking scenarios: the child thread either acquires the lock $l1$ if the boolean condition on variable b holds or acquires the lock $l2$ if the condition does not hold. Suppose that the main thread is going to join the child thread. The main thread, knowing that b holds, can exclude the deadlock scenario that the child thread potentially attempts to acquire the lock $l2$. Hence it is deadlock-free to join the child thread. Note that due to our assumption on data-race freedom, the boolean condition on variable b is consistent in both threads.

4 Verification Rules

Proof rules for forward verification are presented in Fig. 7. They are formalized using Hoare's triples of the form $\{\Phi_{pr}\}P\{\Phi_{po}\}$: given a program P beginning in a state satisfying the pre-condition Φ_{pr}, if it terminates, it will do so in a state satisfying the post-condition Φ_{po}. In the figure, we only focus on key statements that are related to concurrency and lockset: procedure call, fork/join, conditional, and lock operations. In our framework, each program state $\Delta_\mu[(\mathbf{and}\ \Delta_\tau)^*]$ consists of the current state Δ_μ of a main (or parent) thread and a list of final states Δ_τ of child threads. Here final states of child threads refer to post-states of child threads after they finish execution and their delayed formulae that need to be checked at join points. When joined, the post-state of a child thread will be visible and merged into the state of the main thread if its delayed formula is satisfied. For simplicity of presentation, when discussing the rules for fork/join, we present a program state Δ_μ **and** Δ_τ consisting of two threads (a thread main Δ_μ and a child thread Δ_τ). Additionally, because other rules only affect the main thread, it is sufficient to present only state of the main thread Δ_μ.

In order to invoke a procedure call (CALL) in a sequential setting, a main thread should be in a state Δ_μ that can entail the pre-condition Φ_{pr} of the

procedure *pn*. For brevity, we omit the substitutions that link actual and formal parameters of the procedure prior to the entailment. We also omit the treatment of pass-by-ref parameters which can be handled by applying permissions on variables [20, 30]. After the entailment, the main thread subsumes the post-condition Φ_{po} of the procedure into its state. Note that the operator $\wedge_{\{w^*,\mathbf{LS},\mathbf{waitlevel}\}}$ is a "composition with update" operator [28] to capture effects of executing the procedure on its parameters w^*, **LS**, and **waitlevel**.

The auxiliary function *partLS* is used in concurrent settings to partition a formula into a delayed formula γ (which will be "delayed for checking") and a pure formula π. In case of a disjunctive formula, the corresponding delayed formula is also in a disjunctive form. This is to ensure that deadlock-free preconditions on lock acquisition can be more precisely guaranteed when "delayed checking". The auxiliary function *removeLS* removes constraints that are related to lockset and waitlevel because they are irrelevant in concurrent settings. The semantics of *removeLS* is straightforward, hence it is not presented.

The rules for fork and join demonstrate the *delayed lockset checking* technique. A **fork** creates a new thread executing concurrently with the main thread.

$$partLS([\textstyle\bigwedge\omega \,\#\, \bigwedge\psi] \wedge \pi) \stackrel{\text{def}}{=} (\textstyle\bigwedge\psi \wedge \pi_1, \pi_1) \text{ where } \pi_1 := removeLS(\pi)$$
$$partLS(\Phi_1 \vee \Phi_2) \stackrel{\text{def}}{=} (\gamma_1 \vee \gamma_2, \pi_1 \vee \pi_2) \quad \text{where } (\gamma_1,\pi_1) := partLS(\Phi_1) \text{ and } (\gamma_2,\pi_2) := partLS(\Phi_2) \qquad \textbf{AUX}$$
$$partLS(\mu \textbf{ and } \tau) \stackrel{\text{def}}{=} (\pi \textbf{ and } \tau, \gamma) \text{ where } (\gamma,\pi) := partLS(\mu)$$

$$\frac{def(pn) := \mathtt{pn}(w^*) \textbf{ requires } \Phi_{pr} \textbf{ ensures } \Phi_{po};\ \{\ s\ \} \quad \Delta_\mu \vdash \Phi_{pr}}{\{\Delta_\mu\}\ pn(w^*)\ \{\Delta_\mu \wedge_{\{w^*,\mathbf{LS},\mathbf{waitlevel}\}} \Phi_{po}\}} \qquad \textbf{CALL}$$

$$\frac{\begin{array}{c} def(pn) := \mathtt{pn}(w^*) \textbf{ requires } \Phi_{pr} \textbf{ ensures } \Phi_{po};\ \{\ s\ \} \\ (\gamma_{pr},\pi_{pr}) := partLS(\Phi_{pr}) \qquad (\_,\pi_{po}) := partLS(\Phi_{po}) \\ \Delta_\mu \vdash \pi_{pr} \qquad fresh(id) \qquad \Delta^1_\mu := \Delta_\mu \wedge_{\{v\}} v'{=}id \\ \Delta_\tau := \mathbf{thread}{=}id \wedge \gamma_{pr} \rightsquigarrow_{\{w^*\}} \pi_{po} \\ \{\Phi_{pr} \wedge \mathbf{LS}{=}\{\}\}\ s\ \{\Phi_{po} \wedge \mathbf{LS}'{=}\{\}\} \end{array}}{\{\Delta_\mu\}\ v := \mathbf{fork}(pn,w^*)\ \{\Delta^1_\mu \textbf{ and } \Delta_\tau\}} \qquad \textbf{FORK}$$

$$\frac{\Delta_\mu \vdash \gamma_{pr}}{\{\Delta_\mu \textbf{ and } \mathbf{thread}{=}v \wedge \gamma_{pr} \rightsquigarrow_{\{w^*\}} \pi_{po}\}\ \mathbf{join}(v)\ \{\Delta_\mu \wedge_{\{w^*\}} \pi_{po}\}} \qquad \textbf{JOIN}$$

$$\frac{\{\Delta_\mu \wedge b\}\ s_1\ \{\Delta^1_\mu\} \qquad \{\Delta_\mu \wedge \neg b\}\ s_2\ \{\Delta^2_\mu\}}{\{\Delta_\mu\}\ \textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2\ \{\Delta^1_\mu \vee \Delta^2_\mu\}} \qquad \textbf{COND}$$

$$\frac{fresh(l)}{\{\Delta_\mu\}\ \mathtt{lock}\ l = \textbf{new } \mathtt{lock}(v)\ \{\Delta^1_\mu \wedge l{\neq}\mathtt{null} \wedge l.mu{=}v \wedge l \notin \mathbf{LS}'\}} \qquad \textbf{NEWLOCK}$$

$$\frac{\Delta_\mu \vdash \mathbf{waitlevel}{<}l.mu}{\{\Delta_\mu\}\ \mathbf{acquire}(l)\ \{\Delta_\mu \wedge_{\{\mathbf{LS}\}} \mathbf{LS}'{=}\mathbf{LS} \cup \{l\}\}} \qquad \textbf{ACQUIRE}$$

$$\frac{\Delta_\mu \vdash l \in \mathbf{LS}}{\{\Delta_\mu\}\ \mathbf{release}(l)\ \{\Delta_\mu \wedge_{\{\mathbf{LS}\}} \mathbf{LS}'{=}\mathbf{LS}{-}\{l\}\}} \qquad \textbf{RELEASE}$$

Fig. 7. Forward Verification Rules for Concurrency

When forking a new child thread (**FORK**), because lockset and waitlevel are local to each thread, the state of the main thread needs not entail constraints related to waitlevel and lockset in the pre-condition Φ_{pr} of the child thread. However, the main thread should be in a state that can entail the formula π_{pr}. The delayed formula γ_{pr} is delayed for checking at a join point. Afterwards, a new thread Δ_τ with a fresh identifier id carrying the delayed formula γ_{pr} and the post-state π_{po} of the corresponding forked procedure is created. The main thread keeps the identifier of the child thread in its new state Δ^1_μ via the return value v of the **fork** call. Note that constraints related to lockset and waitlevel in the post-condition Φ_{po} are also omitted (resulted in π_{po}) because they are only local to the child thread and are irrelevant to the context of the main thread after the child thread is joined. Lastly, to guarantee the ownership semantics of locks, the **FORK** rule checks if the forked procedure with an empty lockset in its pre-condition will finally end up with an empty lockset in its post-condition. Alternatively, this check could be done during the verification of each forkable procedure without breaking information hiding at call sites.

Joining a child thread with an identifier v (**JOIN**) requires that the state Δ_μ of the main thread must entail the child thread's delayed formula γ_{pr}. The main thread then merges the post-state of the child thread π_{po} into its state and the child thread disappears from the program state after joined.

The rule for conditionals (**COND**) illustrates our support for *precise lockset reasoning*. We capture precise lockset by using disjunction in the post-state of the conditional statement. Together with disjunctive delayed formulae supported by the function *partLS* in **FORK** rule, the use of explicit disjunction in this rule enables more precise reasoning on locksets to ensure deadlock freedom.

Other verification rules are relatively straightforward. The **NEWLOCK** rule creates a new lock l with a locklevel v. Without specifying a locklevel, a lock is assumed to have an arbitrary non-zero locklevel. We assume that locklevel is immutable during a lock's lifetime. The **ACQUIRE** rule ensures that locks are acquired in an increasing oder of locklevels (**waitlevel**$<l.mu$). This additionally implies that $l \notin$**LS** (but not vice versa). After acquiring the lock l, it is added to the thread's lockset **LS**. Reversely, a thread must hold a lock ($l \in$**LS**) in order to release it (**RELEASE**). After releasing the lock l, it is removed from the thread's lockset **LS**. The **ACQUIRE** and **RELEASE** rules respectively ensure that a lock is not acquired or released more than once. The rest of verification rules used in our framework only operate in sequential settings, therefore they are standard as described in [28].

Theorem 1 (Soundness) *Given a program with a set of procedures P^i and their corresponding pre/post-conditions (Φ^i_{pr}/Φ^i_{po}), if our verifier derives a proof for every procedure P^i, i.e. $\{\Phi^i_{pr}\}P^i\{\Phi^i_{po}\}$ is valid, the program is deadlock-free.*

Proof. Intuitively, for each program state, there is a wait-for graph corresponding to it. We prove that a program that has been successfully verified by our framework will never get stuck due to deadlocks, i.e. there does not exist a state whose wait-for graph contains a cycle. The full proof is given in Appendix C. □

5 Implementation and Preliminary Comparison

We have implemented our framework into a prototype tool, called ParaHIP[5]. Currently, ParaHIP can automatically verify different deadlock scenarios and several motivating concurrent programs presented in the literature [11, 14, 15]. In addition, our tool can handle programs with forking of recursive procedures (such as the well-known parallel Fibonacci program) and unbounded number of locks by using shape predicates. We also support intricate nested and non-lexical fork/join concurrency by allowing thread identifiers to be passed between threads. Such a program is outlined in Appendix D.

To demonstrate the expressiveness of our framework, we did a comparison with Chalice [22, 23], a well-known framework for verifying deadlock freedom, in terms of deadlock/deadlock-freedom scenarios that can be proven by the respective frameworks. The benchmark programs cover various scenarios such as double lock acquisition, interactions between thread and lock operations, and unordered locking. One scenario (e.g. double acquisition) is representative of many real-world programs. Therefore, although the scenarios are small, they can be considered as a core benchmark for evaluating expressiveness of deadlock verification systems. The sets of benchmark programs written for both Chalice and ParaHIP are available for online testing in our project website.

The comparison results are presented in Table 1. Compared with Chalice, ParaHIP allows more deadlocks to be prevented and also permits more programs to be declared as deadlock-free. The experimental results were very surprising because Chalice appears unsound. We communicated this issue with Chalice' developers and confirmed that Chalice's technical framework is in-

[5] The tool is available for both online use and download at `http://loris-7.ddns.comp.nus.edu.sg/~project/parahip/`.

Table 1. A comparison between Chalice and ParaHIP. A tick (✓) indicates that the corresponding scenario can be verified correctly by the respective verification framework. A cross (✗) indicates otherwise. A prefix "disj" indicates that the corresponding scenario requires disjunctive formulae to precisely capture different execution branches.

No	Scenario	Chalice	ParaHIP	Comments
1	no-deadlock1	✗	✓	Chalice cannot prove that this program is deadlock-free
2	no-deadlock2	✓	✓	
3	no-deadlock3	✗	✓	Chalice cannot prove that this program is deadlock-free
4	deadlock1	✗	✓	Chalice verifies this deadlock scenario as deadlock-free
5	deadlock2	✓	✓	
6	deadlock3	✓	✓	
7	disj-no-deadlock	✓	✓	
8	disj-deadlock	✗	✓	Chalice verifies this deadlock scenario as deadlock-free
9	ordered-locking	✓	✓	
10	unordered-locking	✓	✓	

deed sound but its implementation does not properly consider programs with interactions between thread and lock operations [26]. Due to space limitations, we refer interested readers to Appendix E and project website for detailed comparison and benchmark programs.

6 Related Work

This section presents related works on specification and verification of deadlock freedom in shared-memory concurrency. Note that our framework currently supports only partial correctness. Hence, we do not consider non-termination due to infinite loops or recursion. Proving (non-)termination [2, 7] and livelock freedom [29] is orthogonal to our framework, and could be separately extended.

In the context of concurrency verification, several recent frameworks have been proposed to reason about programs with non-recursive locks and dynamically-created threads [11, 14], recursive locks [12], and low-level languages [8], all based on separation logic [31]. However, they focus on verifying partial correctness and ignore the presence of deadlocks. Haack et al. [12] use locksets (but not precise locksets) when verifying partial correctness of concurrent programs manipulating Java recursive locks. However, their approach does not ensure deadlock freedom. VERIFAST [15], a state-of-the-art verifier, also ignores deadlocks when verifying correctness of concurrent programs. CHALICE [22, 23], a verification framework based on implicit dynamic frames [32], is capable of preventing deadlocks. Initially, Chalice uses locklevels and is able to prevent deadlocks due to double acquisition and unordered locking [22]. Later development on CHALICE [23] has proposed a technique to prevent deadlocks in programs that use both message passing via channels, and locking. Although it could encode join operations as send/receive over channels, there are programs (such as the program `fork-join-as-send-recv` in our website) where it is impossible for the encoding to find proper levels assigned to the channels for proving deadlock freedom [26]. Our *delayed lockset checking* technique can enable proving deadlock freedom in the presence of interactions between fork/join and acquire/release based on *precise lockset* as an abstraction. Using the technique, we are able to prove more programs deadlock-free than previous work. We also showed how to incorporate locklevels into our technique to form an expressive framework for specifying and verifying deadlock freedom of concurrent programs.

Besides verification frameworks, there are other approaches to detect or prevent deadlocks in concurrent programs. They can be classified into dynamic and static approaches. There are many systems that detect deadlocks dynamically - see [5, 16, 25] to name just a few recent works on this topic. Dynamic systems have the advantage that they can check unannotated programs. However, they cannot guarantee the absence of deadlocks due to insufficient test coverage. Static approaches such as those based on static analysis [27, 34] and type systems [3, 9, 10, 33] can ensure the absence of certain types of deadlocks. These systems have the advantage that fewer annotations are required. However, they tend to be less expressive than specification logics. Type systems such as [3, 33] use locklevels to enforce a locking order while others use lock capabilities [10] and continuation effects [9] to verify programs with no natural ordering on the

locks acquired. Nevertheless, existing systems [3, 9, 10, 33] do not ensure the absence of deadlocks due to interactions between thread and lock operations. It is interesting to apply our delayed lockset checking technique to enhance the capability of these type systems.

Deadlock-freedom has also been studied in other contexts, and notably in the setting of message-passing process algebra [17–19]. The notion of locklevels in our approach is similar to obligation and capability levels in these type systems [17–19]. However, they have only been applied in the context of π-calculus while our framework ensures deadlock freedom for a shared-memory concurrent language with dynamic creation of threads and locks. Although fork/join/acquire/release operations and shared variables could be encoded as send/receive operations over channels, such an encoding would be non-trivial [17, 35].

7 Conclusion

In this paper, we presented an expressive deadlock-freedom verification framework for concurrent programs. A novel delayed lockset checking technique is introduced to cover deadlock scenarios due to interactions between thread and lock operations. We described an abstraction based on precise lockset to support verification for deadlock freedom. We then showed how our technique can be integrated with locklevels to form a formalism for verifying different deadlock scenarios such as those due to double acquisition, interactions between thread and lock operations, and unordered locking. Lastly, we implemented the proposed framework into ParaHIP, a prototype verifier based on separation logic reasoning, for specifying and verifying deadlock freedom and partial correctness of concurrent programs.

Acknowledgement. We thank Peter Müller for his insightful discussions about Chalice, and the anonymous reviewers for comments. This work is supported by MOE Project 2009-T2-1-063.

References

1. NetBSD Problem Report 42900. `http://gnats.netbsd.org/42900`.
2. M. F. Atig, A. Bouajjani, M. Emmi, and A. Lal. Detecting Fair Non-termination in Multithreaded Programs. In *CAV*, pages 210–226, 2012.
3. C. Boyapati, R. Lee, and M. C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, pages 211–230, 2002.
4. D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.
5. Y. Cai and W. K. Chan. MagicFuzzer: Scalable Deadlock Detection for Large-scale Applications. In *ICSE*, pages 606–616, 2012.
6. E. G. Coffman, M. J. Elphick, and A. Shoshani. System Deadlocks. *ACM Computing Surveys*, 3(2):67–78, 1971.
7. B. Cook, A. Podelski, and A. Rybalchenko. Proving Program Termination. *CACM*, 54(5):88–98, 2011.
8. M. Fu, Y. Zhang, and Y. Li. Formal Reasoning about Concurrent Assembly Code with Reentrant Locks. In *TASE*, pages 233–240, 2009.
9. P. Gerakios, N. Papaspyrou, and K. F. Sagonas. A Type and Effect System for Deadlock Avoidance in Low-level Languages. In *TLDI*, pages 15–28, 2011.

10. C. S. Gordon, M. D. Ernst, and D. Grossman. Static Lock Capabilities for Deadlock Freedom. In *TLDI*, pages 67–78, 2012.
11. A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local Reasoning for Storable Locks and Threads. In *APLAS*, pages 19–37, 2007.
12. C. Haack, M. Huisman, and C. Hurlin. Reasoning about Java's Reentrant Locks. In *APLAS*, pages 171–187, Berlin, Heidelberg, 2008.
13. Tony Hoare. The Verifying Compiler: A Grand Challenge for Computing Research. *JACM*, 50:63–69, January 2003.
14. A. Hobor, A. W. Appel, and F. Z. Nardelli. Oracle Semantics for Concurrent Separation Logic. In *ESOP*, pages 353–367, Berlin, Heidelberg, 2008.
15. B. Jacobs and F. Piessens. Expressive Modular Fine-grained Concurrency Specification. In *POPL*, pages 271–282, New York, NY, USA, 2011.
16. P. Joshi, M. Naik, K. Sen, and D. Gay. An Effective Dynamic Analysis for Detecting Generalized Deadlocks. In *FSE*, pages 327–336, 2010.
17. N. Kobayashi. Type-based Information Flow Analysis for the Pi-calculus. *Acta Informatica*, 42(4-5):291–347, 2005.
18. N. Kobayashi. A New Type System for Deadlock-Free Processes. In *CONCUR*, pages 233–247, 2006.
19. N. Kobayashi and D. Sangiorgi. A Hybrid Type System for Lock-Freedom of Mobile Processes. *TOPLAS*, 32(5), 2010.
20. D.K. Le, W.N. Chin, and Y.M. Teo. Variable Permissions for Concurrency Verification. In *ICFEM*. Springer, 2012.
21. E. A. Lee. The Problem with Threads. *Computer*, 39:33–42, May 2006.
22. K. R. M. Leino and P. Müller. A Basis for Verifying Multi-threaded Programs. In *ESOP*, pages 378–393, Berlin, Heidelberg, 2009.
23. K. R. M. Leino, P. Müller, and J. Smans. Deadlock-Free Channels and Locks. In *ESOP*, pages 407–426, 2010.
24. S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from Mistakes: a Comprehensive Study on Real World Concurrency Bug Characteristics. In *ASPLOS*, pages 329–339, New York, NY, USA, 2008.
25. Z. D. Luo, R. Das, and Y. Qi. Multicore SDK: A Practical and Efficient Deadlock Detector for Real-World Applications. In *ICST*, pages 309–318, 2011.
26. P. Müller. Personal communication, March 2013.
27. M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective Static Deadlock Detection. In *ICSE*, pages 386–396, 2009.
28. H.H. Nguyen, C. David, S.C. Qin, and W.N. Chin. Automated Verification of Shape and Size Properties via Separation Logic. In *VMCAI*, Nice, France, 2007.
29. J. Ouaknine, H. Palikareva, A. W. Roscoe, and J. Worrell. Static Livelock Analysis in CSP. In *CONCUR*, pages 389–403, 2011.
30. M. Parkinson, R. Bornat, and C. Calcagno. Variables as Resource in Hoare Logics. In *LICS*, pages 137–146, Washington, DC, USA, 2006.
31. J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*, Copenhagen, Denmark, July 2002.
32. J. Smans, B. Jacobs, and F. Piessens. Implicit Dynamic Frames. *TOPLAS*, 2012.
33. K. Suenaga. Type-Based Deadlock-Freedom Verification for Non-Block-Structured Lock Primitives and Mutable References. In *APLAS*, pages 155–170, 2008.
34. A. Williams, W. Thies, and M. D. Ernst. Static Deadlock Detection for Java Libraries. In *ECOOP*, pages 602–629, 2005.
35. J.M. Wing. FAQ on Pi-calculus. In *Microsoft Internal Memo*, 2002.

A Lockset as an Abstraction

While most previous works [11, 14, 22, 23] use *abstract predicates* for reasoning about concurrent programs manipulating locks, we advocate the use of explicit *locksets*. Though monadic predicates are logically equivalent to sets, they are not always realised as such for two reasons. Firstly, we often avoid the use of negation for predicates, such as $\neg$*Locked(x)*, since such operator may be difficult to implement and may have to rely on the concept of negation as failure, as used in Prolog. Secondly, the application of frame rule [31] makes it more difficult to reason about the absence of a given predicate. Thus it is harder to reason about absence of locks using predicates to avoid deadlocks. In contrast, with lockset, if a callee is going to acquire a non-recursive lock x, it is simpler to check if a given lock is not in the current thread's lockset (denoted by **LS**), by using $x \notin$**LS** in the pre-condition of the callee. Nevertheless, such check can be considered as sound only if the given lockset is *precise* and not an *approximation*, as explained in Section 2.

B Programming Language

In this section, we describe the imperative core language presented in Section 3. Our language supports fork/join concurrency for dynamic thread creation and non-recursive locks. A program consists of a list of procedure declarations *proc*. Each procedure *proc* is annotated with pairs of pre/post-conditions (Φ_{pr}/Φ_{po}) written in our specification logic (Section 3.2). A parameter can be passed by value or by reference (`ref`). The programming language supports primitive types (such as `int`, `bool`, and `void`), and lock type `lock`. Key statements for concurrency consist of fork/join for thread management and lock operations. A **fork** receives a procedure name `pn` and a list of parameters v^*, spawns a new thread executing the procedure, and returns a unique thread identifier as an integer. A **join** requires a thread identifier to join the thread back. `lock` $v =$ **new** `lock`(v) creates a new lock with a ghost argument representing its locklevel. **acquire**(v) and **release**(v) attempt to acquire and respectively release the lock v. The semantics of other program statements (such as procedure calls `pn`(v^*), conditionals, loops, assignments) are standard as can be found in well-known languages such as C/C++.

C Soundness Proof

In this section, we outline a proof to show that the proposed framework guarantees deadlock freedom with respect to the language described in Fig. 5. Deadlocks are well-known and one of the most cited definitions of deadlocks is by Coffman et al. [6]. Four conditions must hold for a deadlock to occur: (1) "mutual exclusion", (2) "no preemption", (3) "wait for", and (4) "circular wait". In our framework, the first three deadlock conditions hold: use of (mutex) locks (condition 1), a lock cannot be preempted until it is released (condition 2), threads may have to wait when acquiring a lock or joining another thread (condition 3), and we ensure deadlock freedom by breaking the "circular wait" (condition 4).

Our proof is inspired by the proof for deadlock freedom made by Leino et al. [23]. In contrast to their proof which focuses on lock operations and channel send/receive, our proof focuses on lock operations and thread fork/join instead. As a reminder, there is a wait-for graph corresponding to each program state. We prove that for each program that has been successfully verified by our framework, there does not exist a state whose wait-for graph contains a cycle.

Definition 1 (Well-formness) *A program is well-formed if the following conditions hold:*

- *Procedure names are unique within a program. Procedure parameters are unique within a procedure. Free variables in the body of a procedure are the procedure parameters.*
- *A normal procedure call or a fork statement mentions only procedure names defined in the program text. The number of actual parameters and formal parameters are equal.*
- *Free variables in pre/post specifications are the procedure parameters plus lockset variable* **LS** *and waitlevel variable* **waitlevel**.

A thread can be in one of three states: *running*, *done*, and *aborted*. Our verification framework ensures that no thread ends up in an *aborted* state. A program state is *non-aborting* if neither of threads are in an *aborted* state. A program state is *final* if all threads are in a *done* state.

Definition 2 (Thread State) *A thread state σ is one of the following states:*

- $\mathbf{run}(s, \Gamma)$ *stating that the thread is running with remaining statement s and environment Γ. For brevity, Γ is assumed to be a partial function from object names to object references and from stack variables to values. Environment Γ resembles stack and heap in programs.*
- **done** *stating that a thread has completed its execution.*
- **aborted** *stating a thread has performed an illegal operation.*

Definition 3 (Program State) *A program state Ψ consists of:*

- *L representing a partial function from locks to locklevels. Thus, $L(o)$ denotes the locklevel of lock o. A lock is already allocated if $o \in dom(L)$.*
- *T representing a set of threads. Each thread is a tuple $(\iota, \sigma, \kappa, ls)$ consisting of thread identifier ι, thread state σ, set of locks κ which the thread intends to acquire since the beginning of its execution, and set of locks ls currently held by the thread.*

Definition 4 (Execution) *Execution of a program starts in the initial program state:* $(\ \emptyset,\ (\_, \mathbf{run}(s, \emptyset), \emptyset, \emptyset)\ )$

Fig. 8 shows the small-step operational semantics. A premise marked with box denotes the fact that threads must block and wait for the premise to become true. For example, a thread can only acquire a lock which is not held by any thread. A premise marked with light gray indicates conditions that need to hold, otherwise the thread has performed an illegal operation and it transitions to an aborted state. For example, a thread will abort if it attempts to release a lock without holding it. Our framework ensures that the premises in light gray hold, i.e. threads cannot transition to aborted states. The rules presented require that a thread starts and completes its execution with an empty lockset.

In Fig. 8, $def(pn)$ denotes the definition of the procedure pn in the program, $eval(e, \Gamma)$ denotes the evaluation of the expression e in the environment Γ, $delayed(\Phi, \Gamma)$ denotes the set of locks that a thread intends to acquire since the beginning of its execution (i.e. the delayed lockset). $delayed(\Phi, \Gamma)$ is defined in Definition 5 based on the thread's pre-condition Φ and an environment Γ. Note that at run-time, we know which disjuncts are satisfiable.

Definition 5 (Delayed Lockset) *Let Φ be a specification (described in Section 3.2) whose free variables are in dom(Γ). The delayed lockset of Φ is defined as follows:*

$$delayed(\Phi_1 \vee \Phi_2, \Gamma) = delayed(\Phi_1, \Gamma) \cup delayed(\Phi_2, \Gamma)$$

$$delayed(\Phi, \Gamma) = \begin{cases} delayed\_aux(\Phi, \Gamma) & \text{if } \Phi \text{ is } SAT \text{ under } \Gamma \\ \emptyset & \text{otherwise} \end{cases}$$

$$delayed\_aux([\textstyle\bigwedge \omega \ \#\ \bigwedge \psi] \wedge \pi, \Gamma) = delayed\_aux(\textstyle\bigwedge \psi, \Gamma)$$
$$delayed\_aux(\psi_1 \wedge \psi_2, \Gamma) = delayed\_aux(\psi_1, \Gamma) \cup delayed\_aux(\psi_2, \Gamma)$$
$$delayed\_aux(x \in \mathbf{LS}, \Gamma) = \{\Gamma(x)\}$$

Definition 6 (Wait-for Graph) *Each program state (L, { $(\iota_1, \sigma_1, \kappa_1, ls_1)$, ..., $(\iota_n, \sigma_n, \kappa_n, ls_n)$ }) forms a directed wait-for graph whose nodes are the threads in the program state. This graph contains an arc from thread $(\iota_{t_1}, \sigma_{t_1}, \kappa_{t_1}, ls_{t_1})$ to thread $(\iota_{t_2}, \sigma_{t_2}, \kappa_{t_2}, ls_{t_2})$ if one of the following conditions holds:*

- *Thread t_1 blocks waiting for thread t_2 to release a lock. In other words, σ_{t_1} is $\mathbf{run}(\mathbf{acquire}(x); s, \Gamma_{t_1})$,$\Gamma_{t_1}(x) \in ls_{t_2}$, and σ_{t_1} cannot go to an aborted state.*
- *Thread t_1 blocks waiting for thread t_2 to terminate. In other words, σ_{t_1} is $\mathbf{run}(\mathbf{join}(\iota_{t_2}); s, \Gamma_{t_1})$, and σ_{t_1} cannot go to an aborted state.*

Each program state Ψ has a corresponding directed wait-for graph. A *deadlock* occurs if the wait-for graph contains a cycle. Theorem 2 states that an arc in the graph between t_1 and t_2 implies that t_1's waitlevel is smaller than t_2's waitlevel or lockset ls_1 of t_1 does not contain the lock that t_2 is waiting to acquire while t_1 is waiting for t_2 at a join point. Theorem 3 states that, for each program state, there is always a thread that is able to make progress. Following from Theorem 3, Theorem 4 states the main soundness theorem for deadlock-freedom.

Theorem 2 (Arc in Wait-for Graph) *If the wait-for graph corresponding to a non-aborting program state has an arc from $(\iota_{t_1}, \sigma_{t_1}, \kappa_{t_1}, ls_{t_1})$ to $(\iota_{t_2}, \sigma_{t_2}, \kappa_{t_2}, ls_{t_2})$, then one of the following properties holds:*

$$\frac{o \notin dom(L) \qquad typeof(o) = \texttt{lock} \qquad \Gamma(w) = level \qquad level{>}0 \qquad \Gamma' = \Gamma[v \mapsto o] \qquad L' = L[o \mapsto level]}{(L, \{(\iota, \mathbf{run}(v = \mathbf{new}\ \texttt{lock}(w); s, \Gamma), \kappa, ls)\} \cup T) \to (L', \{(\iota, \mathbf{run}(s, \Gamma'), \kappa, ls)\} \cup T)}$$

$$(L, \{(\iota, \mathbf{run}(\mathbf{if}\ \mathit{true}\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2; s, \Gamma), \kappa, ls)\} \cup T) \to (L, \{(\iota, \mathbf{run}(s_1; s, \Gamma), \kappa, \emptyset)\} \cup T)$$

$$(L, \{(\iota, \mathbf{run}(\mathbf{if}\ \mathit{false}\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2; s, \Gamma), \kappa, ls)\} \cup T) \to (L, \{(\iota, \mathbf{run}(s_2; s, \Gamma), \kappa, \emptyset)\} \cup T)$$

$$\frac{eval(e, \Gamma) = b}{(L, \{(\iota, \mathbf{run}(\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2; s, \Gamma), \kappa, ls)\} \cup T) \to (L, \{(\iota, \mathbf{run}(\mathbf{if}\ b\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2; s, \Gamma), \kappa, \emptyset)\} \cup T)}$$

$$\frac{def(pn) := \texttt{pn}(w_1, \ldots, w_n)\ \mathbf{requires}\ \Phi_{pr}\ \mathbf{ensures}\ \Phi_{po};\ \{\ s_1\ \} \qquad s_1' = [v_1/w_1, \ldots, v_n/w_n]s_1}{(L, \{(\iota, \mathbf{run}(pn(v_1, \ldots, v_n); s, \Gamma), \kappa, ls)\} \cup T) \to (L, \{(\iota, \mathbf{run}(s_1'; s, \Gamma), \kappa, ls)\} \cup T)}$$

$$\frac{def(pn) := \texttt{pn}(w_1, \ldots, w_n)\ \mathbf{requires}\ \Phi_{pr}\ \mathbf{ensures}\ \Phi_{po};\ \{\ s_1\ \} \qquad \forall i \in \{1, \ldots, n\} \bullet \Gamma(v_i) = o_i \qquad \mathit{fresh}(\iota_1) \qquad \Gamma' = [v \mapsto \iota_1] \qquad \Gamma_1 = [w_1 \mapsto o_1, \ldots, w_n \mapsto o_n] \qquad \kappa_1 = delayed(\Phi_{pr}, \Gamma')}{(L, \{(\iota, \mathbf{run}(v = \mathbf{fork}(pn, v_1, \ldots, v_n); s, \Gamma'), \kappa, ls)\} \cup T) \to (L, \{(\iota, \mathbf{run}(s, \Gamma), \kappa, ls)\} \cup \{(\iota_1, \mathbf{run}(s_1, \Gamma_1), \kappa_1, \emptyset)\} \cup T)}$$

$$\frac{\boxed{\exists(\iota_1, (\mathbf{done}, \Gamma), \_, \_) \in T \bullet \Gamma(v) = \iota_1}}{(L, \{(\iota, \mathbf{run}(\mathbf{join}(v); s, \Gamma), \kappa, ls)\} \cup T) \to (L, \{(\iota, \mathbf{run}(s, \Gamma), \kappa, ls)\} \cup T)}$$

$$\frac{\Gamma(x) = o \qquad \boxed{\forall(\_, \_, \_, ls_t) \in T \bullet o \notin ls_t} \qquad ls' = ls \cup \{o\} \qquad o \notin ls \qquad \forall l \in dom(L) \bullet l \in ls \Rightarrow L(l) < L(o)}{(L, \{(\iota, \mathbf{run}(\mathbf{acquire}(x); s, \Gamma), \kappa, ls)\} \cup T) \to (L, \{(\iota, \mathbf{run}(s, \Gamma), \kappa, ls')\} \cup T)}$$

$$\frac{\Gamma(x) = o \qquad o \in ls \qquad ls' = ls - \{o\}}{(L, \{(\iota, \mathbf{run}(\mathbf{release}(x); s, \Gamma), \kappa, ls)\} \cup T) \to (L, \{(\iota, \mathbf{run}(s, \Gamma), \kappa, ls')\} \cup T)}$$

$$\frac{ls = \emptyset}{(L, \{(\iota, \mathbf{run}(\texttt{skip}, \Gamma), \kappa, ls)\} \cup T) \to (L, \{(\iota, \mathbf{done}, \kappa, \emptyset)\} \cup T)}$$

Fig. 8. Small-step operational semantics for well-formed programs

- $max\{L(o) \mid o \in ls_{t1}\} < max\{L(o) \mid o \in ls_{t2}\}$
- σ_{t_1} *equals* $\mathbf{run}(\mathbf{join}(\iota_{t_2}); s, \Gamma_{t_1})$, *and* $ls_{t_1} \cap \kappa_{t_2} = \emptyset$

Proof. Since there is an arc from t_1 to t_2, t_1 cannot go into an aborted state. We consider two cases:

- **Acquire**. If the first statement of t_1 is $\mathbf{acquire}(x)$ and Γ is t_1's environment with $\Gamma(x) = o$, then it follows from the premise that

$$\forall l \in dom(L) \bullet l \in ls_{t_1} \Rightarrow L(l) < L(o) \quad \text{or} \quad max\{L(l) \mid l \in ls_{t_1}\} < L(o)$$

Because $o \in ls_{t_2}$, this implies $L(o) \leq max\{L(l) \mid l \in ls_{t_2}\}$. The first property holds.

– **Join**. The delayed lockset checking ensures that t_1 is not holding any locks that t_2 is going to acquire, that is, $ls_{t_1} \cap \kappa_{t_2} = \emptyset$. The second property holds. □

Theorem 3 (Deadlock Freedom) *If a program state Ψ is non-final and non-aborting, then Ψ is not stuck.*

Proof. By proving that there is always a thread that is able to make progress, i.e. the graph corresponding to Ψ contains a non-final thread t that has no outgoing arc. If the first statement s_1 of t is neither **acquire** nor **join**, then t can make progress. If s_1 is an **acquire**(x), then no other thread holds the lock x (otherwise t would have an outgoing arc). Hence, t can acquire x. If s_1 is **join**(id), the thread with identifier id has completed its execution (otherwise t would have an outgoing arc). Therefore, t can make progress. □

Theorem 4 (Soundness) *Given a program with a set of procedures P^i and their corresponding pre/post-conditions (Φ^i_{pr}/Φ^i_{po}), if our verifier derives a proof for every procedure P^i, i.e. $\{\Phi^i_{pr}\}P^i\{\Phi^i_{po}\}$ is valid, the program is deadlock-free.*

Proof. It follows from Theorem 3 that each program that has been successully verified by our framework never gets stuck due to deadlocks. □

D Verification Example

In this section, we apply the rules presented in Section 4 to verification of a deadlock-free program with intricate non-lexical fork/join operations (Fig. 9). A main thread, after forking a child thread `thread1`, creates another thread `thread2` and allows `thread2` to join with `thread1`. The fork and join operations of `thread1` is non-lexical because they are in different scopes (or more precisely in different threads). This makes the program intricate and difficult to verify it as deadlock-free. As a small experiment, we gave this small program to three colleagues in our research lab and asked them whether it is deadlock-free. Interestingly, all of them pondered on the program in more than fifteen minutes but neither of them could figure out that the program is deadlock-free in the first attempt.

We apply our verification framework to this program and prove that it is indeed deadlock-free. In order to support this kind of non-lexical fork/join concurrency, our framework allows threads to be transferred between callers and callees during the entailments. The formal treatment of the entailments is orthogonal to the contributions of this paper and is omitted for simplicity. Intuitively, when forking the thread `thread2`, the main thread transferred the thread with thread identifier `tid1` to `thread2`. Afterwards, the main thread adds `thread2` into its set of concurrent threads. Because a thread can only be transferred from one

```
void thread1(lock l)
  requires [waitlevel<l.mu # l∉LS] ∧ l≠null
  ensures LS′=LS;
{ acquire(l); release(l); }

void thread2(lock l,int tid1)
  requires l∉LS ∧ l≠null
           and thread=tid1 ∧ l∉LS′ ∧ l≠null ⇝{l,tid1} true
  ensures LS′=LS;
{
  // { l∉LS ∧ l≠null ∧ LS′=LS
  //   and thread=tid1 ∧ l∉LS′ ∧ l≠null ⇝{l} true }
  join(tid1);/*CHECK, ok*/
  // { l∉LS ∧ l≠null ∧ LS′=LS }
}

void main()
  requires LS={}  ensures LS′={};
{
  lock l = new lock();
  // { ∃v1 · l≠null ∧ l.mu=v1 ∧ v1>0 ∧ l∉LS′ ∧ LS′={} }
  int tid1 = fork(thread1,l);/*DELAY*/
  // { ∃v1,v2 · l≠null ∧ l.mu=v1 ∧ v1>0 ∧ tid1=v2 ∧ LS′={}
  //   and thread=v2 ∧ l∉LS′ ∧ l≠null ⇝{l} true }
  acquire(l);
  // { ∃v1,v2 · l≠null ∧ l.mu=v1 ∧ v1>0 ∧ tid1=v2 ∧ LS′={l}
  //   and thread=v2 ∧ l∉LS′ ∧ l≠null ⇝{l} true }
  int tid2 = fork(thread2,l,tid1);/*DELAY*/
  // { ∃v1,v2,v3 · l≠null ∧ l.mu=v1 ∧ v1>0 ∧ tid1=v2 ∧ tid2=v3 ∧ v2≠v3 ∧ LS′={l}
  //   and thread=v3 ∧ l∉LS′ ∧ l≠null ⇝{l,tid1} true }
  release(l);
  // { ∃v1,v2,v3 · l≠null ∧ l.mu=v1 ∧ v1>0 ∧ tid1=v2 ∧ tid2=v3 ∧ v2≠v3 ∧ LS′={}
  //   and thread=v3 ∧ l∉LS′ ∧ l≠null ⇝{l,tid1} true }
  join(tid2);/*CHECK, ok*/
  // { ∃v1,v2,v3· l≠null∧l.mu=v1∧v1>0∧tid1=v2∧tid2=v3∧v2≠v3∧LS′={} }
}
```

Fig. 9. Proof outline of a deadlock-free program with non-lexical fork/join concurrency

thread to another thread, our semantics ensures that only one thread can perform a join. The transfer of threads together with the delayed lockset checking technique guarantee deadlock freedom in the presence of non-lexical fork/join concurrency.

E Comparison

Table 1 shows the comparison between Chalice and ParaHIP. Compared with Chalice, ParaHIP allows more deadlocks to be prevented (more sound) and also permits more programs to be declared as deadlock-free (more complete).

Specifically, Chalice is unable to correctly verify 4 out of 10 scenarios that express intricate interactions between thread and lock operations. The last column in the table briefly explains the reason behind. The sets of benchmark programs written for both Chalice and ParaHIP are available for aniline's testing in our project website[6].

The experimental results were very surprising because Chalice appears unsound. We communicated this issue with Chalice' developers and confirmed that Chalice's technical framework is indeed sound but its implementation does not consider programs with interactions between thread and lock operations [26]. Hence, the question to investigate is whether Chalice could be extended to handle those scenarios? To the best of our knowledge, Chalice technical framework could, under the hood, encode fork/join as send/receive over channels, assign levels to the channels, and require that threads acquire locks and wait on channels in a strictly increasing order of (locks' and channels') levels (Section 4.4 of [23]). With this encoding, Chalice becomes sound and it can automatically eliminate the false negatives in the programs `deadlock1` and `disj-deadlock`. However, it still does not correctly verify the programs `no-deadlock1` and `no-deadlock3` as deadlock-free. To be more expressive, Chalice could be extended to allow programmers to explicitly annotate appropriate levels to thread identifiers and require that threads acquire locks and join threads in a strictly increasing order of (locks' and thread identifiers') levels [26]. With this extended help from programmers, Chalice could correctly verify all programs in Table 1. However, there are still programs (such as the program `fork-join-as-send-recv` in our project website) where it is impossible to find appropriate levels to assign to the thread identifiers for proving deadlock freedom [26]. That program can be verified as deadlock-free in our framework without requiring extended help from programmers. In summary, compared with Chalice, our framework is more expressive in handling interactions between fork/join and lock operations. It advocates the use of precise locksets and introduces the delayed lockset checking technique to more expressively prove deadlock freedom.

[6] `http://loris-7.ddns.comp.nus.edu.sg/~project/parahip/`.