Automated Verification of CountDownLatch

Wei-Ngan Chin[1] Ton Chanh Le[1] Shengchao Qin[2]

[1]National University of Singapore
[2]Teesside University

Abstract. The `CountDownLatch` (CDL) is a versatile concurrency mechanism that was first introduced in Java 5, and is also being adopted into C++ and C#. Its usage allows one or more threads to exchange resources and synchronize by waiting for some tasks to be completed before others can proceed. In this paper, we propose a new framework for verifying the correctness of concurrent applications that use CDLs. Our framework is built on top of two existing mechanisms, *concurrent abstract predicate* and *fictional separation logic*, with some enhancements such as *borrowed heap* and *thread local abstraction*. In addition, we propose a new inconsistency detection mechanism based on waits-for relation to guarantee *deadlock freedom*. Prior concurrency verification works have mostly focused on *data-race freedom*. As a practical proof of concept, we have implemented this new specification and verification mechanism for CDL in a new tool, called HIPCAP, on top of an existing HIP verifier. We have used this new tool to successfully verify various use cases for CDL.

1 Introduction

One of the most popular techniques for reasoning about concurrent programs is separation logic [26, 30]. Originally, separation logic was designed to verify heap-manipulating sequential programs, with the ability to express non-aliasing in the heap [30]. Separation logic was also extended to verify shared-memory concurrent programs, e.g. concurrent separation logic [26], where ownerships of heap objects are considered as *resource*, which can be shared and transferred among concurrent threads. Using fractional permissions [2, 4], one can express full ownerships for exclusive write accesses and partial ownerships for concurrent read accesses. Ownerships of stack variables could also be considered as resource and treated in the same way as heap objects [3].

Most existing solutions to verify the correctness of concurrent programs have focused on simpler concurrency primitives, such as binary semaphores [26], locks [10, 11] and first-class threads [21, 9]. Lately, some solutions are beginning to emerge for more complex concurrency mechanisms, such as barriers [13] and channels [7] (using only grant/wait). The former used multiple pre/post specifications for every thread at each barrier point to capture resource exchanges, but this requires some non-local reasoning over each set of multiple pre/posts. The latter relied on higher-order concurrent abstract predicates (HOCAP) to provide abstract predicates that are stable in the presence of interfering actions

from concurrent threads. However, with the unrestricted use of higher-order predicates, some concurrent abstract predicates proposed in [7] were unsound.

Moreover, we are not aware of any formal verification solution that have been applied to the more versatile `CountDownLatch` where multiple threads could be involved in data exchanges. One new challenge is the need to handle two different kinds of threads namely, (i) those that perform non-blocking countdown operations, and (ii) those that perform blocking await operations. We resolve them through two new mechanisms (i) distinct abstract predicates can semantically distinguish producers from consumers for concurrency synchronization mechanisms; (ii) *inconsistency lemma* that can detect deadlocks by tracking when one CDL command is to be completed before another. On correctness guarantees, we show how to ensure both *race-freedom* and *deadlock-freedom*. Most existing solutions on concurrency verification have focused on only race-freedom, whilst some solutions [22, 20] have been proposed for verifying deadlock-freedom, mostly in the context of mutex locks and channels.

Our paper makes the following technical contributions:

- We propose the first formal verification for `CountDownLatch`. We use two resource predicates and one counting predicate to soundly track resource synchronization for our concurrent programs using `CountDownLatch`(s).
- We provide a modular solution to the count down mechanism by supporting a *thread-local* abstraction on top of the usual *global* view on its shared counter. While this can be viewed as an instance of fictional separation logic [32], our use of thread-local abstraction goes beyond that by allowing the interference effects of parallel threads to be modularly and precisely aggregated.
- We provide interpretations for our abstract predicates. This involves a novel use of *septraction* operator to capture the notion of *borrowed* heap.
- We highlight two desired concurrency properties (i) race-freedom, and (ii) deadlock-freedom, that can be ensured by formally verifying our concurrent abstract predicates for `CountDownLatch`. A novel feature is the use of *inconsistency lemmas* to help detect concurrency errors, after they occur.
- We provide a prototype verifier for `CountDownLatch`. We use it to verify several applications and a library implementation of the `CountDownLatch`.

2 Motivation for `CountDownLatch`

The `CountDownLatch` protocol is a synchronization mechanism that allows one or more threads to wait until a certain number of operations are completed by other threads. A `CountDownLatch` instance is initialized with a non-negative count. A call of the `await` primitive blocks the current thread until the count of its latch reaches zero. At that point, all waiting threads are released and any subsequent invocation of `await` returns immediately. Note that the count of each latch is only decreased by one, down to zero for each invocation of the `countDown` primitive and cannot be reset. The `CountDownLatch` is a versatile mechanism that can be used for some non-trivial synchronization patterns.

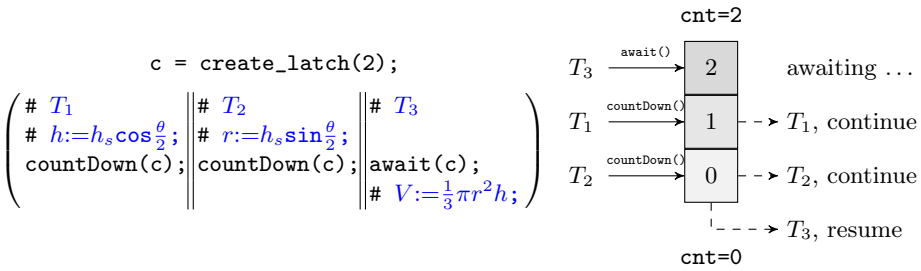


Fig. 1. A use-case of `CountDownLatch`

Our first use-case of `CountDownLatch` is when a thread must wait for other threads to complete their work, so that all resources needed for its computation are available. As a simple example, we show a program which calculates the volume V of a circular cone given its slant height h_s and aperture θ. In this program, the height h and the radius r of the cone are concurrently computed by two threads T_1 and T_2. When both their computations have completed, these threads separately invoke the `countDown` method to inform the thread T_3, via awaiting on the latch to reach zero, to continue its computation for V based on h and r, as illustrated in Fig. 1.

In the second use-case, we leverage `CountDownLatch` to implement a copyless multi-cast communication pattern, where a single send is being awaited by multiple receivers. As an example, let us assume that resource `P*Q`[1] is being sent by a thread, which are awaited by two other threads, receiving `P` and `Q`, respectively. This copyless multi-cast communication can be modeled by the following concurrent program with a `CountDownLatch` initialized to 1.

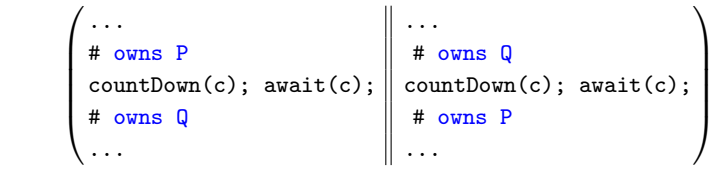
```
c = create_latch(1);

( # send P*Q      ||  await(c);        ||  await(c);
  countDown(c);  ||  # receive P      ||  # receive Q
  ...            ||  ...              ||  ...          )
```

The same pattern can be used to coordinate the starting of several threads, in which the `countDown` action of the first thread is a starting signal for other awaiting threads to start at the same time.

Last but not least, a barrier synchronization can be implemented using `CountDownLatch`. As another example, consider two threads, that own `P` and `Q` respectively, but wish to exchange their respective resources at a barrier. We may model this scenario by using one `countDown` immediately followed by `await`, at each of the barrier point in the two threads, as follows:

```
c = create_latch(2);
```

[1] `P*Q` is a separation logic formula formed by separation conjunction `*` [15, 30].



As seen with these examples, the communication patterns here are non-trivial and we shall see how resource predicates can help us perform formal reasoning on `CountDownLatch`.

3 Concurrent Abstract Predicates for `CountDownLatch`

In this section, we propose a set of concurrent abstract predicates that can be used to formally model the `CountDownLatch`. Apart from allowing its resources to be more precisely tracked,another key challenge faced by the `CountDownLatch` concurrency protocol is the ability to support thread modular reasoning for its shared counter. We introduce two *resource* predicates `LatchIn(c,P)` and `LatchOut(c,P)` to help track resource `P` precisely, and one counting predicate `CNT(c,n)` to help track countdowns. Each set of three concurrent abstract predicates is created to model a `CountDownLatch`, as follows:

```
CountDownLatch create_latch(n) with P
   requires n>0
   ensures LatchIn(res,P)*LatchOut(res,P)*CNT(res,n)_1;
   requires n=0
   ensures CNT(res,-1)_1;
```

Note the variable `res` denotes the return value of the method. We used two pre/post specifications to describe this constructor. In case `n=0`, the latch cannot be used for concurrency synchronization and is simply denoted by a final state of $\texttt{CNT}(\texttt{res},\texttt{-1})_\epsilon$ to denote an expired latch whose count is definitely 0. (We attach fractional permission ϵ to `CNT` to help track sharing on the `CountDownLatch`. Such a scenario would allow the latch itself to be de-allocated, where desired.) In case `n>0`, the resource predicates, `LatchIn(c,P)` and `LatchOut(c,P)`, are used to model the inflow and outflow, respectively, of some resource `P` that are being exchanged by the `CountDownLatch`. Specifically, the predicate `LatchIn(c,P)` shall be used for the *consumption* of resource `P` into the `CountDownLatch` (at `countDown` call), while the predicate `LatchOut(c,P)` shall be used to model the *production* of `P` from the `CountDownLatch` (at each `await` call), as shown below.

```
void countDown(CountDownLatch i)
  requires LatchIn(i,P)*P*CNT(i,n)_ε∧n>0
  ensures CNT(i,n-1)_ε;
  requires CNT(i,-1)_ε
  ensures CNT(i,-1)_ε;
```

```
void await(CountDownLatch i)
  requires LatchOut(i,P)*CNT(i,0)_ε
  ensures P*CNT(i,-1)_ε;
  requires CNT(i,-1)_ε
  ensures CNT(i,-1)_ε;
```

The higher-order formula `P` denotes the logical resource added by our specifications for `CountDownLatch` to support race-free resource exchanges, as the

underlying specification of `CountDownLatch` is focused exclusively on its count-down counter and the blocking effects on threads from `await` calls. The predicate $\texttt{CNT}(\texttt{c},\texttt{n})$ does not capture any resources, but is used to provide an abstract view of the counter inside `CountDownLatch`. A novel feature of this `CNT` predicate is that it could be used to support both a global view and a thread local view. Note that $\texttt{CNT}(\texttt{c},\texttt{n})_\epsilon \wedge \texttt{n} \geq 0$ gives a thread-local view of the counter with a value of at least `n`. As a special case, $\texttt{CNT}(\texttt{c},0)_\epsilon$ denotes a shared counter that is at least `0`, while $\texttt{CNT}(\texttt{c},\texttt{-1})_\epsilon$ denotes a shared counter that is definitely `0` (i.e. the counter has reached its final state with value `0`). In order to support resource consumption by the first pre/post specification of `countDown` method, we require $\texttt{CNT}(\texttt{c},\texttt{n})_\epsilon \wedge \texttt{n} > 0$ to ensure race-free synchronization.

The following normalization rules show how multiple `CNT` instances are combined prior to each formal reasoning step to provide a sound view of the shared counter. The first rule shows idempotence on the final state of $\texttt{CNT}(\texttt{c},\texttt{-1})$. The second rule combines multiple `CNT` instances, where possible. The third rule allows resource trapped in each latch to be released at its final state. This rule helps preserve (or release) unconsumed resources from each expired latch.

$$[\textsc{norm}-1] : \texttt{CNT}(\texttt{c},\texttt{n})_{\epsilon_1} * \texttt{CNT}(\texttt{c},\texttt{-1})_{\epsilon_1} \wedge \texttt{n} \leq 0 \longrightarrow \texttt{CNT}(\texttt{c},\texttt{-1})_{\epsilon_1+\epsilon_2}$$
$$[\textsc{norm}-2] : \texttt{CNT}(\texttt{c},\texttt{n1})_{\epsilon_1} * \texttt{CNT}(\texttt{c},\texttt{n2})_{\epsilon_2} \wedge \texttt{n}=\texttt{n1}+\texttt{n2} \wedge \texttt{n1},\texttt{n2} \geq 0 \longrightarrow \texttt{CNT}(\texttt{c},\texttt{n})_{\epsilon_1+\epsilon_2}$$
$$[\textsc{norm}-3] : \texttt{LatchOut}(\texttt{c},\texttt{P}) * \texttt{CNT}(\texttt{c},\texttt{-1})_\epsilon \longrightarrow \texttt{CNT}(\texttt{c},\texttt{-1})_\epsilon * \texttt{P}$$

To support distribution to multiple concurrent threads, we provide a set of splitting lemmas for our abstract predicates, as follows:

$$[\textsc{split}-1] : \texttt{LatchOut}(\texttt{i},\texttt{P}*\texttt{Q}) \longrightarrow \texttt{LatchOut}(\texttt{i},\texttt{P}) * \texttt{LatchOut}(\texttt{i},\texttt{Q})$$
$$[\textsc{split}-2] : \texttt{LatchIn}(\texttt{i},\texttt{P}*\texttt{Q}) \longrightarrow \texttt{LatchIn}(\texttt{i},\texttt{P}) * \texttt{LatchIn}(\texttt{i},\texttt{Q})$$
$$[\textsc{split}-3] : \texttt{CNT}(\texttt{c},\texttt{n})_{\epsilon_1+\epsilon_2} \wedge \texttt{n1},\texttt{n2} \geq 0 \wedge \texttt{n}=\texttt{n1}+\texttt{n2} \longrightarrow \texttt{CNT}(\texttt{c},\texttt{n1})_{\epsilon_1} * \texttt{CNT}(\texttt{c},\texttt{n2})_{\epsilon_2}$$

Our splitting process is guided by the pre-condition expected for each concurrent thread. This split occurs at the start of fork/par operations. For example, if an `await` call is in a thread, our split will try to ensure that a suitable thread-local state, say $\texttt{LatchOut}(\texttt{i},\texttt{P}) * \texttt{CNT}(\texttt{i},0)$, is passed to this thread.

We illustrate an example in Fig. 2 where splitting lemmas allow relevant abstract predicates to be made available for the modular verification of each thread. For simplicity in presentation, we shall omit fractional permissions for the `CNT` predicate in the rest of this paper. As such permissions must be tracked precisely, they are always conserved by lemmas and pre/post specifications.

```
                    c = create_latch(2) with P*Q;
                  # LatchOut(c,P*Q)*LatchIn(c,P*Q)*CNT(c,2)
    # LatchOut(c,P*Q)*LatchIn(c,P)*LatchIn(c,Q)*CNT(c,0)*CNT(c,1)*CNT(c,1)
```

···	··· create P ···	··· create Q ···
# LatchOut(c,P*Q)*CNT(c,0)	# P*LatchIn(c,P)*CNT(c,1)	# Q*LatchIn(c,Q)*CNT(c,1)
await(c);	countDown(c);	countDown(c);
# P*Q*CNT(c,-1)	# CNT(c,0)	# CNT(c,0)
··· use P*Q ···	···	···

Fig. 2. A `CountDownLatch` with two threads counting down

3.1 Ensuring Race Freedom

In the CountDownLatch protocol, we expect all countDown calls that are executed concurrently to the await calls be completed before the latter. This is important to ensure *race-freedom*, since resources that are generated from countDown operations must be completed (or used), before they are transferred to threads that are blocked by the await calls. In order to ensure this, we require the precondition of countDown method to be either CNT(c, n)∧n>0 or CNT(c, -1), but never CNT(c, 0). Thus, each violation on the pre-condition of countDown[2] would be signaled as a potential race problem. Another situation where race problem can occur is when resources that are required by await calls are not being synchronized by any countDown call. Such an example is illustrated in Fig. 3.

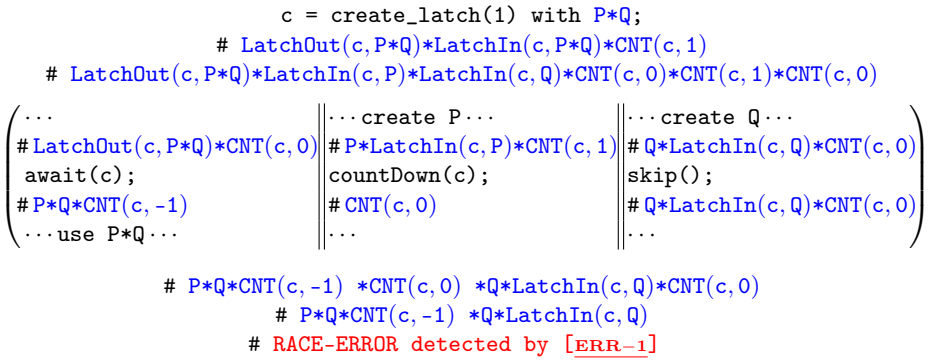


Fig. 3. A CountDownLatch with a race error

The third thread has the resource Q generated there but was not being synchronized by any countDown call. As a consequence, Q was not properly transferred to its corresponding await call. We propose to detect such potential race violations through the following inconsistency lemma:

[ERR−1]: LatchIn(c, P)*CNT(c, -1) ⟶ RACE-ERROR

The formula LatchIn(c, P)*CNT(c, -1) denotes a contradiction. The former predicate requires the shared counter to be non-zero, while latter predicate is in a final state with value 0 (see Sec 3.3). Such contradictions are manifestations of some concurrency synchronization errors. We identify this as a race problem.

3.2 Ensuring Deadlock Freedom

Deadlock may occur when blocking operations, such as await, are invoked and then could wait forever, e.g., due to the shared counter never reaching zero.

For a single CountDownLatch, this deadlock error can be signified by the following lemma:

[ERR−2]: CNT(c, a)*CNT(c, -1)∧a>0 ⟶ DEADLOCK-ERROR

Here, CNT(c, a)∧a>0 denotes a counter value of at least 1, while CNT(c, −1) denotes

[2] This violation occurs when skip() in the third thread of the example in Fig. 3 is replaced by countDown(c).

a final counter with value 0. Such a contradiction is a manifestation of a deadlock error which arose from the unreachability on precondition $\texttt{CNT}(c,0) \vee \texttt{CNT}(c,\texttt{-1})$ that we impose on the `await` method. A simple example of this deadlock scenario (omitting `Latch` predicates) is shown in Fig 4. This deadlock error occurs due to the first thread not invoking a sufficient number of `countDown` calls. This lack of `countDown` calls led to an error scenario when the conflicting states from the two threads are being joined.

```
        c = create_latch(2);
  # CNT(c,2) ⟶ CNT(c,2)*CNT(c,0)
(# CNT(c,2)      ||  # CNT(c,0)   )
(countDown(c);   ||  await(c);    ) ;
(# CNT(c,1)      ||  # CNT(c,-1)  )
        # CNT(c,1)*CNT(c,-1)
# DEADLOCK-ERROR detected by [ERR-2]
```

Fig. 4. An intra deadlock scenario

For multiple latches, we will need to track a wait-for graph [33], to help detect deadlocks, where possible. Let us introduce a $\texttt{WAIT}\,(\texttt{S})_\epsilon$ relation that is tracked inter-procedurally with its wait-for graph `S` and its permission ϵ. Whenever we have a complete view of the waits-for relation (with full permission, denoted as $\texttt{WAIT}\,(\texttt{S})_1$), we can always reset each acyclic wait-for graph to `{}`, as follows: $[\textbf{WAIT-1}]$: $\texttt{WAIT}\,(\texttt{S})_1 \wedge \neg\texttt{isCyclic}(\texttt{S}) \longrightarrow \texttt{WAIT}\,(\{\})_1$
Note that `isCyclic(S)` returns `true` when the wait-for graph `S` contains a cycle. We add a wait-for arc via this lemma:

$[\textbf{WAIT-2}]$: $\texttt{CNT}(c_1,a)*\texttt{CNT}(c_2,-1)*\texttt{WAIT}\,(\texttt{S})_\epsilon \wedge a{>}0 \longrightarrow \texttt{CNT}(c_1,a)*\texttt{CNT}(c_2,-1) \wedge a{>}0 * \texttt{WAIT}\,(\texttt{S} \cup \{c_2 \rightarrow c_1\})_\epsilon$

An arc $c_2 \rightarrow c_1$ is added into the `WAIT` relation to indicate that (the counting down on) c_2 will be completed before the `CountDownLatch` c_1 or in other words, c_1 is waiting for c_2 to complete. The wait-for predicates are split/normalized by:

$[\textbf{WAIT-3}]$: $\texttt{WAIT}\,(\texttt{S}_1)_{\epsilon_1} * \texttt{WAIT}\,(\texttt{S}_2)_{\epsilon_2} \longleftrightarrow \texttt{WAIT}\,(\texttt{S}_1 \cup \texttt{S}_2)_{\epsilon_1+\epsilon_2}$

We normalize where possible, and split the `WAIT` relation at fork/par locations. Moreover, any occurrence of a cycle in the wait-for graph is immediately detected as a potential deadlock by the following inconsistency lemma:

$[\textbf{ERR-3}]$: $\texttt{WAIT}\,(\texttt{S})_\epsilon \wedge \texttt{isCyclic(S)} \longrightarrow \texttt{DEADLOCK-ERROR}$

An example of deadlock detection for multiple latches is shown in Fig 5 where the deadlock is detected by the cycle in $\texttt{WAIT}\,(\{c2 \rightarrow c1, c1 \rightarrow c2\})$.

```
  c1 = create_latch(1); c2 = create_latch(1);
    # WAIT({})1*CNT(c1,1) * CNT(c2,1) ⟶
# CNT(c1,1) * CNT(c2,0) * CNT(c2,1) * CNT(c1,0)
(# CNT(c1,1)*CNT(c2,0)      ||  # CNT(c2,1)*CNT(c1,0)     )
(await(c2);                 ||  await(c1);                 )
(# CNT(c1,1)*CNT(c2,-1)     ||  # CNT(c2,1)*CNT(c1,-1)    )
(#   *WAIT({c2→c1})ε1       ||  #   *WAIT({c1→c2})ε2       )
(countDown(c1);             ||  countDown(c2);             )
(# CNT(c1,0)*CNT(c2,-1)     ||  # CNT(c2,0)*CNT(c1,-1)    )
(#   *WAIT({c2→c1})ε1       ||  #   *WAIT({c1→c2})ε2       )
#   CNT(c1,-1)*CNT(c2,-1)*WAIT({c2→c1, c1→c2})1
    # DEADLOCK-ERROR detected by [ERR-3]
```

Fig. 5. An example of inter-latch deadlock

Though deadlock detection for locks and channels have been proposed before [22], there are at least two novel ideas in our proposal. Firstly, we have now shown how to formally ensure deadlock-freedom for the more complex `CountDownLatch` protocol. Secondly, we have achieved this through the use of inconsistency lem-

mas. The wait-for arcs added are used for inconsistency detection, since each arc $\mathtt{c}_2 \rightarrow \mathtt{c}_1$ denotes a strict completion ordering $\mathtt{c}_2 < \mathtt{c}_1$ (for a pair of latches) that is to be expected from its concurrent execution. Any cycle from such an accumulated waits-for ordering denotes potential unsatisfiability.

3.3 Interpretations for Abstract Predicates

We have introduced three concurrent abstract predicates, namely $\mathtt{LatchIn}(\mathtt{c},\mathtt{P})$, $\mathtt{LatchOut}(\mathtt{c},\mathtt{P})$ and $\mathtt{CNT}(\mathtt{c},\mathtt{n})$, for the specification of `CountDownLatch`. The first two predicates are concerned with managing the flows of resource `P` through its `CountDownLatch`. We can provide the following interpretations for them:

$\mathtt{LatchOut}(\mathtt{i},\mathtt{P}) \stackrel{\text{def}}{=} \boxed{\mathtt{i}\mapsto\mathtt{0}} \longrightarrow \mathtt{P}$

$\mathtt{LatchIn}(\mathtt{i},\mathtt{P}) \stackrel{\text{def}}{=} (\mathtt{P} -\!\circledast \mathtt{emp}) * [\mathtt{DEC}]_\epsilon * \boxed{\mathtt{i}\mapsto\mathtt{m}} \wedge \mathtt{m}{>}\mathtt{0}$

Each formula $\boxed{\mathtt{i}\mapsto\mathtt{m}}$ denotes a shared global location, while $\mathtt{i}\mapsto\mathtt{m}$ denotes a fully-owned heap (local) location. Predicate $\mathtt{LatchOut}(\mathtt{i},\mathtt{P})$ is itself a *producer* of resource `P` that is released once shared global counter becomes `0`. For the $\mathtt{LatchIn}(\mathtt{i},\mathtt{P})$ predicate, we use $\mathtt{P} -\!\circledast \mathtt{emp}$ with a *septraction* operator $-\!\circledast$ ([36]) to capture the *consumption* of resource `P` into the `CountDownLatch` via its `countDown(i)` method. Unlike the septraction operator in [36] which works with *real* heaps, our formula $\mathtt{P} -\!\circledast \mathtt{emp}$ is an extension to capture the concept of *virtual* heap that denotes a borrowing of heap `P`. For example, $(\mathtt{x}\mapsto\_) -\!\circledast \mathtt{emp}$ is a borrowing of $(\mathtt{x}\mapsto\_)$ such that $\forall \mathtt{Q}\cdot(\mathtt{Q} -\!\circledast \mathtt{emp}) * \mathtt{Q} = \mathtt{emp}$. This simple (but novel) concept allows us to capture notion of resource flows through the `CountDownLatch`. The $\mathtt{LatchIn}(\mathtt{i},\mathtt{P})$ predicate also captures a partial permission for `DEC` action that causes its shared global counter $\boxed{\mathtt{i}\mapsto\mathtt{m}}$ to be decreased by `1`, as captured by:

$\mathtt{DEC}: \boxed{\mathtt{i}\mapsto\mathtt{n}} \wedge \mathtt{n}{>}\mathtt{0} \rightsquigarrow \boxed{\mathtt{i}\mapsto\mathtt{n}\text{-}\mathtt{1}}$

To support local reasoning with updates, we propose a *thread-local* view for global counters based on fictional separation logic [17]. We introduce a new *thread-local* formula $\{\!|\mathtt{i}\mapsto\mathtt{n}|\!\}$ to denote a shared counter whose value is at least `n`. Such a thread-local abstraction is related to its global counter by the property:

$\{\!|\mathtt{i}\mapsto\mathtt{n}|\!\} \longrightarrow \boxed{\mathtt{i}\mapsto\mathtt{m}} \wedge \mathtt{m}{\geq}\mathtt{n}{\geq}\mathtt{0}$

While the formula $\boxed{\mathtt{i}\mapsto\mathtt{m}}$ provides a precise view of some shared global location, the thread-local formula $\{\!|\mathtt{i}\mapsto\mathtt{m}|\!\}_\epsilon$ provides a fictional thread-local view of the same counter that could be separately updated by each thread. This applies even if fractional permission is being imposed for the thread-local predicate, as long as such predicates can be shown to be stable.

To support concurrency, we provide combine/split operations that can be used to precisely handle multiple thread-local states from concurrent threads.

$\{\!|\mathtt{i}\mapsto\mathtt{n}|\!\}_{\epsilon_1+\epsilon_2} \wedge \mathtt{a},\mathtt{b}{\geq}\mathtt{0} \wedge \mathtt{n}{=}\mathtt{a}{+}\mathtt{b} \longleftrightarrow \{\!|\mathtt{i}\mapsto\mathtt{a}|\!\}_{\epsilon_1} * \{\!|\mathtt{i}\mapsto\mathtt{b}|\!\}_{\epsilon_2}$

Contrast this to the global view of a shared counter with trivial property:

$\boxed{\mathtt{i}\mapsto\mathtt{a}} * \boxed{\mathtt{i}\mapsto\mathtt{b}} \longleftrightarrow \boxed{\mathtt{i}\mapsto\mathtt{a}} \wedge \mathtt{a}{=}\mathtt{b}$

With these two views, we provide the following intepretation for $\mathtt{CNT}(\mathtt{i},\mathtt{n})$:

$\mathtt{CNT}(\mathtt{i},\mathtt{n}) \stackrel{\text{def}}{=} \{\!|\mathtt{i}\mapsto\mathtt{n}|\!\} \wedge \mathtt{n}{\geq}\mathtt{0} \vee \boxed{\mathtt{i}\mapsto\mathtt{0}} \wedge \mathtt{n}{=}\text{-}\mathtt{1}$

To confirm soundness, we must determine that all three concurrent abstract predicates are stable in the presence of interferring actions. Our use of $\{\!|\mathtt{i}\mapsto\mathtt{n}|\!\}$

remains stable since it is based on fictional separation logic, whose effect is not affected by other concurrent threads. The shared global state $\boxed{\texttt{i}\mapsto\texttt{0}}$ is stable, since `DEC` operation does not modify such a final state of the shared counter. The condition $\boxed{\texttt{i}\mapsto\texttt{m}}\wedge\texttt{m}>\texttt{0}$ of the shared global state in $\texttt{LatchIn}(\texttt{i},\texttt{P})$ is stable since it is always used with $\texttt{CNT}(\texttt{i},\texttt{n})$ in the pre-condition of `countDown` method, which results in $\boxed{\texttt{i}\mapsto\texttt{m}}\wedge\texttt{m}>\texttt{0}*\{\!|\texttt{i}\mapsto\texttt{n}|\!\}\wedge\texttt{n}>\texttt{0}$. By itself the global view $\boxed{\texttt{i}\mapsto\texttt{m}}\wedge\texttt{m}>\texttt{0}$ would not be stable in the presence of concurrent $[\texttt{DEC}]_\epsilon$. However, the combined state $\boxed{\texttt{i}\mapsto\texttt{m}}\wedge\texttt{m}>\texttt{0}*\{\!|\texttt{i}\mapsto\texttt{n}|\!\}\wedge\texttt{n}>\texttt{0}$ remains stable since the thread-local view of $\{\!|\texttt{i}\mapsto\texttt{n}|\!\}\wedge\texttt{n}>\texttt{0}$ is not affected by $[\texttt{DEC}]_\epsilon$ operations from other threads. Thus, $\boxed{\texttt{i}\mapsto\texttt{m}}\wedge\texttt{m}>\texttt{0}$ always hold since $\texttt{m}\geq\texttt{n}$. Lastly, $\texttt{LatchOut}(\texttt{i},\texttt{P})$ is also stable since it only depends on a stable global formula $\boxed{\texttt{i}\mapsto\texttt{0}}$ with final value `0`.

Comparing with Fictional Separation Logic Our use of thread-local abstraction $\{\!|\texttt{i}\mapsto\texttt{n}|\!\}_1$ (equivalent to $\boxed{\texttt{i}\mapsto\texttt{n}}$) is an orthogonal enhancement for fictional separation logic [17]. Though fictional abstraction can be used to reason separately about updates to a shared resource amongst concurrent threads, this abstraction is not truly *thread-local* since its split/aggregate rule does not allow effects from *strong-updates* to be *precisely* propagated amongst concurrent threads. For example, consider the monotonic (increasing) counter from [28]. Though fictional abstraction has the following split/aggregate rule to duplicate or merge the predicate $\texttt{MC}(\texttt{c},\texttt{i})$ of a monotonic counter `c`

$\forall\texttt{i}\leq\texttt{j}\cdot\texttt{MC}(\texttt{c},\texttt{j})_{\texttt{e1}+\texttt{e2}}\longleftrightarrow\texttt{MC}(\texttt{c},\texttt{j})_{\texttt{e1}}*\texttt{MC}(\texttt{c},\texttt{i})_{\texttt{e2}}$

we do not consider it to be thread-local since each predicate in this rule may not maintain a precise view of the shared global counter. Therefore, any update on a predicate via local reasoning may cause other copies to become weaker. To make this abstraction thread-local, we will need to utilize the following conversion:

$\texttt{MC}(\texttt{c},\texttt{n})_1\wedge\texttt{j}\leq\texttt{n}\longleftrightarrow\{\!|\texttt{c}\mapsto\texttt{n}|\!\}_1*\texttt{MC}'(\texttt{c},\texttt{j})_1$

Here, $\{\!|\texttt{c}\mapsto\texttt{n}|\!\}$ is our precise thread-local abstraction, while `MC'(c, j)` is the monotonic counter proposed from [17]. Adding such a thread-local abstraction permits strong updates to be precisely tracked across concurrent threads.

4 Formalism of Language and Logic

$$
\begin{array}{rcl}
\mathit{Prog} & ::= & \overline{\mathit{datat}}\ \ \overline{\mathit{proc}} \\
\mathit{datat} & ::= & \textbf{data}\ \texttt{C}\ \{\ \overline{t\ f}\ \} \\
\mathit{proc} & ::= & t\ pn(\overline{t\ v})\ \ \overline{\mathit{spec}}\ \{\ e\ \} \\
\mathit{spec} & ::= & \texttt{requires}\ \Phi_{pr}\ \texttt{ensures}\ \Phi_{po}; \\
t & ::= & \texttt{void} \mid \texttt{int} \mid \texttt{bool} \mid \texttt{CountDownLatch} \mid \texttt{C} \\
e & ::= & v \mid v.f \mid k \mid \texttt{new}\ \texttt{C}(\overline{v}) \mid e_1; e_2 \mid e_1 \| e_2 \mid \langle\, e \rangle \\
 & & \texttt{create\_latch}(n)\ \texttt{with}\ \kappa\wedge\pi \mid \texttt{countDown}(v) \mid \\
 & & \texttt{await}(v) \mid pn(\overline{v}) \mid \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \ldots
\end{array}
$$

Fig. 6. Core Language with `CountDownLatch`

We use the core language in Fig. 6 to formalise our reasoning with `CountDownLatch`. A program consists of data declarations ($\overline{\mathit{datat}}$), and procedure declarations ($\overline{\mathit{proc}}$). Each procedure declaration is annotated with pairs of pre/post-conditions (Φ_{pr}/Φ_{po}). A (countdown) latch, created by `create_latch`(n), is initialized to a given count n and can be passed in with a (logical) resource $\kappa\wedge\pi$. A `countDown(v)` operation decrements the count of latch `v`. An `await(v)` operation blocks until the count

of latch v reaches zero, after which all waiting threads are released and any subsequent invocation of `await` returns immediately. The operation $\langle e \rangle$ denotes an atomic action. Other program constructs are standard as can be found in the mainstream languages.

$$
\begin{array}{rrl}
\text{FA Pred.} & \mathit{rpred} & ::= \mathtt{pred\ R(self}, \bar{\mathtt{V}}, \bar{\mathtt{v}}) \ [\equiv\ \Phi]\ [\mathbf{inv}\ \pi] \\
\text{Action} & \mathit{act} & ::= \mathtt{action\ I}\ \equiv\ \overline{\iota_1 \wedge \pi_1 \rightsquigarrow \iota_2 \wedge \pi_2} \\
\text{Disj. formula} & \Phi & ::= \bigvee(\exists \bar{\mathtt{v}} \cdot \eta * \kappa \wedge \pi) \\
\text{Non-Resource} & \eta & ::= [\mathtt{I}]_\xi \mid \mathtt{WAIT}\ (\{\overline{v_1 \rightarrow v_2}\})_\xi \mid \boxed{\mathtt{v} \mapsto \mathtt{C}(\bar{\mathtt{v}})} \mid \eta_1 * \eta_2 \\
\text{Sep. formula} & \kappa & ::= \iota \mid \mathtt{V} \mid \mathtt{R}(\mathtt{v}, \overline{\Phi_{\mathtt{f}}}, \bar{\mathtt{v}}) \mid \kappa_1 * \kappa_2 \\
\text{Simple heap} & \iota & ::= \mathtt{emp} \mid \mathtt{v} \mapsto \mathtt{C}(\bar{\mathtt{v}}) \mid \{\!|\mathtt{v} \mapsto \mathtt{C}(\bar{\mathtt{v}})|\!\} \mid \iota_1 * \iota_2 \\
\text{Perms} & \xi & ::= \epsilon \mid 1 \\
\text{Pure formula} & \pi & ::= \alpha \mid \pi_1 \wedge \pi_2 \mid \pi_1 \vee \pi_2 \mid \neg \pi \mid \exists \mathtt{v} \cdot \pi \mid \forall \mathtt{v} \cdot \pi \\
\text{Arith. formula} & \alpha & ::= \alpha_1^t = \alpha_2^t \mid \alpha_1^t \neq \alpha_2^t \mid \alpha_1^t < \alpha_2^t \mid \alpha_1^t \leq \alpha_2^t \\
\text{Arith. term} & \alpha^t & ::= \mathtt{k} \mid \mathtt{v} \mid \mathtt{k} \times \alpha^t \mid \alpha_1^t + \alpha_2^t \mid -\alpha^t \\
\multicolumn{3}{c}{k \in \textit{integer constants} \quad \mathtt{v} \in \textit{variables}, \bar{\mathtt{v}} \equiv \mathtt{v}_1, .., \mathtt{v}_n \quad \mathtt{C} \in \textit{data names}} \\
\multicolumn{3}{c}{\mathtt{V} \in \textit{resource variables} \quad \mathtt{R} \in \textit{resource pred. names} \quad \epsilon \in (0, 1]}
\end{array}
$$

Fig. 7. Core Specification Language

Fig. 7 shows our specification language for concurrent programs supporting `CountDownLatch`'s abstract resource predicates. A classical separation logic formula Φ is in disjunctive normal form. Each disjunct in Φ consists of formulae η, κ and π. A pure formula π includes standard equality/inequality and Presburger arithmetic. π could also be extended to include other constraints such as set constraints. The non-resource formula η may comprise action permission assertions $[\mathtt{I}]_\xi$, (permission annotated) `WAIT` relations, or global views on shared states $\boxed{\mathtt{v} \mapsto \mathtt{C}(\bar{\mathtt{v}})}$. The heap formula κ can be formed by simple heaps ι, resource variables `V`, resource predicate instances $\mathtt{R}(\mathtt{x}, \overline{\Phi_{\mathtt{f}}}, \bar{\mathtt{v}})$, or via separation conjunction `*` ([15, 30]). Note that a resource variable `V` is a place holder for a formula $\Phi_{\mathtt{f}}$, used in a resource predicate declaration, while a resource predicate instance $\mathtt{R}(\mathtt{x}, \overline{\Phi_{\mathtt{f}}}, \bar{\mathtt{v}})$ encapsulates resources $(\overline{\Phi_{\mathtt{f}}})$ that are accessible via `x`. For precision reasons, these resources are restricted to local entities that can be tracked precisely, such as heap nodes or abstract predicates, but must not include global shared locations or `WAIT` relations. A simple heap ι is formed by data nodes $\mathtt{v} \mapsto \mathtt{C}(\bar{\mathtt{v}})$ (which can also appear in a thread-local view $\{\!|\mathtt{v} \mapsto \mathtt{C}(\bar{\mathtt{v}})|\!\}$).

To allow us to focus on the essential issues, we include limited support on permissions (namely on actions and `WAIT` relations) in our logic, but this aspect can be relaxed to support more features of CSL [1]. Furthermore, instances of `WAIT` relations are only permitted in pre/post specifications (and not in predicate definitions). This is to allow every potential race/deadlock errors to be detected by our lemmas, whenever it might occur. Note also that, different from CAP [6] where each action is annotated with a region, for simplicity we do not explicitly mention regions in our action definition and the shared region to be updated by an action can be recovered from the root pointers in the action specification.

5 Automated Verification

Our verification system is built on top of entailment checking:

$$\Delta_A \vdash_E \Delta_C \rightsquigarrow (\mathcal{D}, \Delta_R)$$

This entailment checks if antecedent Δ_A is precise enough to imply consequent Δ_C, and computes the residue Δ_R for the next program state and resource bindings $\mathcal{D}$. $\mathcal{D}$ is a set of pairs $(\mathtt{V}, \Phi_\mathtt{V})$ where $\mathtt{V}$ is a resource variable with its definition $\Phi_\mathtt{V}$. E is a set of existentially quantified variables from the consequent. This entailment procedure is used for pre-condition and post-condition checking. It is also used by assertion checking during automated verification.

In this section, we focus on the main entailment rules for manipulating resource predicates. The rest of the entailment rules, such as those for manipulating normal data predicates or those for combining/normalizing resource predicates using lemmas introduced in Sec 3, are given in Appendix A. These other rules are adapted from prior works [25, 24] by additionally propagating the bindings $\mathcal{D}$. Note that for simplicity, we omit fractional permissions from the predicates.

Resource Predicate Matching. Matching of two resource predicates ([RP−MATCH]) is the key of our approach, i.e. it allows resource predicates to be split and identifies necessary resource bindings for later entailments. For simplicity, we illustrate the rule with at most one resource per predicate and we can handle predicates with multiple resources by splitting them.

$$\frac{\begin{array}{c}[\textbf{RP}-\textbf{MATCH}]\\ \rho = [\bar{\mathtt{v}}_1/\bar{\mathtt{v}}_2] \quad (\mathtt{V}, \Phi_3, \Delta, \mathtt{b}) = addVar(\mathtt{R}(\mathtt{x}, \rho(\Phi_2), \bar{\mathtt{v}}_1))\\ \Phi_1 \vdash^{\delta}_{E\cup\{\mathtt{V}\}} \Phi_3 \rightsquigarrow (\mathcal{D}, \rho_1)\\ \Delta_1 = subst(\mathcal{D}, \kappa_1\wedge\pi_1) \quad \Delta_2 = subst(\mathcal{D}, \Delta) \quad \Delta_3 = subst(\mathcal{D}, \rho(\kappa_2\wedge\pi_2))\\ \Delta_1 * \Delta_2 \wedge freeEqn(\rho\cup\rho_1, E) \vdash_{E-\{\bar{\mathtt{v}}_2\}} \Delta_3 \rightsquigarrow (\mathcal{D}_1, \Delta_4)\\ \mathcal{D}_2 = \mathtt{if\ b\ then}\ \mathcal{D}_1\ \mathtt{else}\ \mathcal{D}\cup\mathcal{D}_1\end{array}}{\mathtt{R}(\mathtt{x}, \Phi_1, \bar{\mathtt{v}}_1) * \kappa_1\wedge\pi_1 \vdash_E \mathtt{R}(\mathtt{x}, \Phi_2, \bar{\mathtt{v}}_2) * \kappa_2\wedge\pi_2 \rightsquigarrow (\mathcal{D}_2, \Delta_4)}$$

$$\begin{array}{l} addVar(\mathtt{R}(\mathtt{x}, \mathtt{V}, \bar{\mathtt{v}})) \quad \stackrel{\text{def}}{=} (\mathtt{V}, \mathtt{V}, \mathtt{emp}, \mathtt{false})\\ addVar(\mathtt{R}(\mathtt{x}, \kappa\wedge\pi, \bar{\mathtt{v}})) \stackrel{\text{def}}{=} (\mathtt{V}, \kappa * \mathtt{V}\wedge\pi, \mathtt{R}(\mathtt{x}, \mathtt{V}, \bar{\mathtt{v}}), \mathtt{true}),\ fresh\ \mathtt{V}\end{array}$$

$$\frac{[\textbf{RP}-\textbf{UNIFY}]\quad \mathcal{D}, \rho = unify(\Phi_1, \Phi_2)}{\Phi_1 \vdash^{\delta}_E \Phi_2 \rightsquigarrow (\mathcal{D}, \rho)} \qquad \frac{[\textbf{RP}-\textbf{INST}]\quad \mathtt{V} \in E}{\Phi \vdash^{\delta}_E \mathtt{V} \rightsquigarrow (\{(\mathtt{V}, \Phi)\}, \emptyset)}$$

In the above [RP−MATCH] rule, we first apply the substitution ρ to unify the components of the corresponding resource predicates in two sides of the entailment. We then resort to an *addVar* method which either uses an existing $\mathtt{V}$ (from the substituted RHS resource argument $\rho(\Phi_2)$) for binding, or adds a fresh variable to the RHS to facilitate the implicit splitting of resource predicates (based on [SPLIT−1] and [SPLIT−2] rules). After that, we invoke a special *resource* entailment $\Phi_1 \vdash^{\delta}_{E\cup\{\mathtt{V}\}} \Phi_3 \rightsquigarrow (\mathcal{D}, \rho_1)$ for resource predicate $\mathtt{R}$, where the resource variable $\mathtt{V}$ is added as an existential variable to discover the resource bindings $\mathcal{D}$ which maps resource variables to resource predicates (if any). Note that if $\mathtt{V}$ is

a fresh variable, such a binding about V will not be kept in the final result. We also use two auxiliary functions: (i) *subst* for substituting a discovered resource variable by its corresponding definition and (ii) *freeEqn* for transferring certain equations (for existential variables from the consequent) to the antecedent for subsequent entailments.

Our resource entailment discovers resource bindings $\mathcal{D}$ based on classic reasoning (without any frame residue) via the rules [**RP−UNIFY**] and [**RP−INST**]. Given two resource formulae Φ_1 and Φ_2, we first check if they are exact heaps by the rule [**RP−UNIFY**] with unification where ρ is the bindings of first-order variables. If there is any resource variable in the consequent formula, we instantiate it by the rule [**RP−INST**] after all common heaps in both sides are unified. An example of the combination of matching ([**RP−MATCH**]), followed resource binding ([**RP−INST**]) is:

$$\texttt{LatchIn(c, x}\mapsto\texttt{cell}(v_1)\texttt{)} \;\vdash\; \texttt{LatchIn(c, V)} \rightsquigarrow (\{(\texttt{V}, \texttt{x}\mapsto\texttt{cell}(v_1))\},\;\; \texttt{emp})$$

In many cases, resource predicates are not matched but are rather split. In these cases, a resource predicate with remaining resources (i.e. added by *addVar*) is returned. The predicate will be then added into the antecedent in the rule [**RP−MATCH**]. For example:

$$\texttt{LatchOut(c, x}\mapsto\texttt{cell}(v_1)\texttt{*y}\mapsto\texttt{cell}(v_2)\texttt{)}$$
$$\vdash\;\; \texttt{LatchOut(c, x}\mapsto\texttt{cell}(v_3)\texttt{)} \;\rightsquigarrow\; (\emptyset, \texttt{LatchOut(c, y}\mapsto\texttt{cell}(v_2)\texttt{)} \wedge v_1{=}v_3)$$

In order to provide the most precise program states, we always perform normalization after each reasoning step. This normalization is performed with the help of ([**RP−COMBINE**]) in Appendix A.

6 Prototype Implementation

We demonstrate the feasibility of our approach in supporting `CountDownLatch` via concurrent abstract predicates by implementing it on top of HIP which originally works for only sequential programs. In contrast, besides verifying functional correctness, data-race freedom, deadlock freedom, our new implementation (called HIPCAP) is now capable of also verifying `CountDownLatch` and beyond through its support for concurrent abstract predicates, shared local abstraction and wait-for relation. Our HIPCAP prototype implementation and a set of verified programs (involving `CountDownLatch` and other concurrency mechanisms) are available for both online use and download.[3]

7 Related Work and Conclusion

Traditional works on concurrency verification such as Owicki-Gries [27] and Rely/Guarantee reasoning [18] are focused on using controlled interference to

[3] The URL is at `http://loris-7.ddns.comp.nus.edu.sg/~project/hipcap/`.

formally reason between each process against those that execute in the background. Subsequent work on concurrency reasoning [26] are based mostly on race-freedom for concurrent processes by allowing processes to share read accesses on some locations, whilst having exclusive write accesses on others. These works were conducted in the presence of thread fork operations [12, 9], and threads and locks [10, 16, 22], and use reasoning machineries, such as RGSep [36], LRG [8], CAP [6], and Views [5], but have not considered more advanced concurrency synchronization, such as `CountDownLatch`. The closest to our work is a recent work on barrier synchronization by Hobor and Gherghina [13] which requires extra pre/post specifications for each thread that are involved in each barrier synchronization to allow resources to be exchanged. This approach requires global inter-thread reasoning to ensure that resources are preserved during each barrier synchronization. In contrast, our use of flow-aware concurrent predicates (for `CountDownLatch`) rely on only modular reasoning and would need a single set of higher-order specifications for its concurrency primitives. Moreover, we also ensure race-freedom and deadlock-freedom.

Our resource predicates are based on Concurrent Abstract Predicates (CAP) [6, 7, 35, 34]. The basic idea behind CAP [6] was to provide an abstraction of possible interferences from concurrently running threads, by partitioning the state into regions with protocols governing how the state in each region is allowed to evolve. Dodds et al. [7] introduced a higher-order variant of CAP to give a generic specification for a library for deterministic parallelism, making explicit use of nested region assertions and higher-order protocols. Svendsen et al. [35] presented a logic called Higher Order Concurrent Abstract Predicates (HOCAP), allowing clients to refine the generic specifications of concurrent data structures. HOCAP uses higher order (predicative) protocols to allow clients to transfer ownership of additional resources to shared data structures.

More recently, there have been a number of modern concurrency logics such as iCAP [34], Iris [19], FCSL[23][31] and CoLoSL [29]. As an improved version of [35], iCAP [34] allows the use of impredicative protocols parameterised on arbitrary predicates and supports modular reasoning about layered and recursive abstractions. Iris [19] combines partial commutative monoids (PCMs) and invariants. Both Iris and iCAP leverage the idea of view shifts, originated by Jacobs and Piessens [16]. FCSL [23, 31], incorporates a uniform concurrency model, based on state-transition systems and PCMs, so as to build proofs about concurrent libraries in a thread-local, compositional way. CoLoSL [29] allows each thread to be verified with respect to its partial subjective view of the global shared state, and uses overlapping conjunction [14] to reconcile the permissions and capabilities, residing in the shared state between concurrent threads. Compared with these general frameworks, our abstract predicates are resource-specific as they explicitly track resources that flow into and out of their abstractions and allow resources to be flexibly split and transferred across procedure and thread boundaries. To deal with the shared counter mechanism, we have now proposed a thread-local abstraction which makes it much more precise and yet simpler to ensure stability of our concurrent abstract predicates. We have also used

inconsistency lemma support to ensure deadlock freedom and race-freedom, and have used resource predicates to help ensure resource preservation – desirable properties that were not properly addressed by prior work on CAP. In another research direction, deadlock avoidance by verification were recently investigated for locks and channels [22, 20]. In comparison, we have provided a new approach based on inconsistency lemma and wait-for set to ensure deadlock freedom.

In conclusion, we have proposed a framework to validate the correctness of concurrent programs using `CountDownLatch`. We showed how to ensure *race-freedom* and *deadlock-freedom*. To the best of our knowledge, this is the first proposal on formal verification for the correctness of concurrent programs using `CountDownLatch`. We have made use of *concurrent abstract predicates* to precisely track resources that are exchanged via `CountDownLatch`. Our proposal allows tracked resources to be re-distributed, in support of sharing and synchronization amongst a group of concurrent threads. We have also proposed inconsistency lemmas to assist with deadlock and race detection. We have followed the approach of [6] for verifying the correctness of an implementation for `CountDownLatch`, but requires two new concepts (i) borrowed heap via $\texttt{P}-\circledast\texttt{emp}$ whereby $\forall\texttt{Q}\cdot(\texttt{Q}-\circledast\texttt{emp})*\texttt{Q} = \texttt{emp}$, and (ii) thread-local abstraction for precise tracking of shared counters. Lastly, proof of soundness of our verification framework is still being developed. It is based the Views framework [5] for modular concurrency reasoning, but have to be extended to cater to several new features, namely (i) inconsistency lemmas (ii) borrowed heap (iii) shared local abstractions, and (iv) deadlock freedom guarantee.

Acknowledgement: We gratefully acknowledge Duy-Khanh Le who highlighted this `CountDownLatch` problem to us and helped with our initial formulation based on flow-aware predicates..

References

1. A. Amighi, C. Haack, M. Huisman, and C. Hurlin. Permission-Based Separation Logic for Multithreaded Java Programs. *CoRR*, abs/1411.0851, 2014.
2. R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission Accounting in Separation Logic. In *POPL*, 2005.
3. R. Bornat, C. Calcagno, and H. Yang. Variables as Resource in Separation Logic. *ENTCS*, 155, 2006.
4. J. Boyland. Checking Interference with Fractional Permissions. In *SAS*, 2003.
5. T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: Compositional Reasoning for Concurrent programs. In *POPL*, 2013.
6. T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP*, 2010.
7. M. Dodds, S. Jagannathan, and M. J. Parkinson. Modular reasoning for deterministic parallelism. In *POPL*, 2011.
8. X. Feng. Local Rely-Guarantee Reasoning. In *POPL*, 2009.
9. X. Feng and Z. Shao. Modular Verification of Concurrent Assembly Code with Dynamic Thread Creation and Termination. In *ICFP*, 2005.
10. A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local Reasoning for Storable Locks and Threads. In *APLAS*, 2007.

11. C. Haack, M. Huisman, and C. Hurlin. Reasoning about Java's Reentrant Locks. In *APLAS*, 2008.
12. A. Hobor. *Oracle Semantics*. PhD thesis, Princeton University, 2008.
13. A. Hobor and C. Gherghina. Barriers in Concurrent Separation Logic: Now With Tool Support! *LMCS*, 8(2), 2012.
14. A. Hobor and J. Villard. The ramifications of sharing in data structures. In *POPL*, 2013.
15. S. S. Ishtiaq and P. W. O'Hearn. BI as an Assertion Language for Mutable Data Structures. In *POPL*, 2001.
16. B. Jacobs and F. Piessens. Expressive Modular Fine-grained Concurrency Specification. In *POPL*, 2011.
17. J. B. Jensen and L. Birkedal. Fictional Separation Logic. In *ESOP*, 2012.
18. C. B. Jones. Specification and Design of (Parallel) Programs. In *IFIP Congress*, 1983.
19. R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, 2015.
20. D.-K. Le, W.-N. Chin, and Y. M. Teo. An Expressive Framework for Verifying Deadlock Freedom. In *ATVA*, 2013.
21. K. R. M. Leino and P. Müller. A Basis for Verifying Multi-threaded Programs. In *ESOP*, 2009.
22. K. R. M. Leino, P. Müller, and J. Smans. Deadlock-Free Channels and Locks. In *ESOP*, 2010.
23. A. Nanevski, R. Ley-Wild, I. Sergey, and G. A. Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *ESOP*, pages 290–310, 2014.
24. H. H. Nguyen and W.-N. Chin. Enhancing program verification with lemmas. In *CAV*, 2008.
25. H. H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated Verification of Shape and Size Properties via Separation Logic. In *VMCAI*, 2007.
26. P. W. O'Hearn. Resources, Concurrency and Local Reasoning. In *CONCUR*, 2004.
27. S. S. Owicki and D. Gries. Verifying Properties of Parallel Programs: an Axiomatic Approach. *CACM*, 19(5), 1976.
28. A. Pilkiewicz and F. Pottier. The essence of monotonic state. In *TLDI*, 2011.
29. A. Raad, J. Villard, and P. Gardner. CoLoSL: Concurrent Local Subjective Logic. In *ESOP*, 2015.
30. J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*, 2002.
31. I. Sergey, A. Nanevski, and A. Banerjee. Mechanized verification of fine-grained concurrent programs. In *PLDI*, 2015.
32. F. Sieczkowski, K. Svendsen, L. Birkedal, and J. Pichon-Pharabod. A separation logic for fictional sequential consistency. In *ESOP*, 2015.
33. A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating system concepts*. Wiley, 2013.
34. K. Svendsen and L. Birkedal. Impredicative concurrent abstract predicates. In *ESOP*, 2014.
35. K. Svendsen, L. Birkedal, and M. J. Parkinson. Modular Reasoning about Separation of Concurrent Data Structures. In *ESOP*, 2013.
36. V. Vafeiadis and M. J. Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR*, 2007.

A Additional Entailment Rules

Additional entailment rules are given in Fig 8.

$$\frac{[\mathbf{EX-L}]}{\mathit{fresh}\ \mathtt{w} \quad [\mathtt{w}/\mathtt{v}]\Delta_1 \vdash_E \Delta_2 \rightsquigarrow (\mathcal{D}, \Delta)}{\exists \mathtt{v} \cdot \Delta_1 \vdash_E \Delta_2 \rightsquigarrow (\mathcal{D}, \Delta)}$$

$$\frac{[\mathbf{EX-R}]}{\mathit{fresh}\ \mathtt{w} \quad \Delta_1 \vdash_{E \cup \mathtt{w}} [\mathtt{w}/\mathtt{v}]\Delta_2 \rightsquigarrow (\mathcal{D}, \Delta_3) \quad \Delta \stackrel{\text{def}}{=} \exists \mathtt{w} \cdot \Delta_3}{\Delta_1 \vdash_E \exists \mathtt{v} \cdot \Delta_2 \rightsquigarrow (\mathcal{D}, \Delta)}$$

$$\frac{[\mathbf{MATCH}] \quad \rho = [\bar{\mathtt{v}}_1/\bar{\mathtt{v}}_2] \quad \kappa_1 \wedge \pi_1 \wedge \mathit{freeEqn}(\rho, E) \vdash_{E - \{\bar{\mathtt{v}}_2\}} \rho(\kappa_2 \wedge \pi_2) \rightsquigarrow (\mathcal{D}, \Delta)}{\mathtt{x} \mapsto C(\bar{\mathtt{v}}_1) * \kappa_1 \wedge \pi_1 \vdash_E \mathtt{x} \mapsto C(\bar{\mathtt{v}}_2) * \kappa_2 \wedge \pi_2 \rightsquigarrow (\mathcal{D}, \Delta)}$$

$$\frac{\begin{array}{c}[\mathbf{RP-COMBINE}] \\ \iota * \kappa \wedge \pi \longrightarrow \kappa' \in \mathcal{L} \qquad \rho = \mathit{match}(\iota, \mathtt{R}(\mathtt{x}, \overline{\Phi_{\mathtt{f}}}^1, \bar{\mathtt{v}}_1)) \\ \mathtt{R}(\mathtt{x}, \overline{\Phi_{\mathtt{f}}}^1, \bar{\mathtt{v}}_1) * \kappa_1 \wedge \pi_1 \vdash_E \rho(\kappa \wedge \pi) \rightsquigarrow (\mathcal{D}_1, \Delta_1) \\ \rho(\kappa') * \Delta_1 \vdash_E \kappa_2 \wedge \pi_2 \rightsquigarrow (\mathcal{D}_2, \Delta)\end{array}}{\mathtt{R}(\mathtt{x}, \overline{\Phi_{\mathtt{f}}}^1, \bar{\mathtt{v}}_1) * \kappa_1 \wedge \pi_1 \vdash_E \kappa_2 \wedge \pi_2 \rightsquigarrow (\mathcal{D}_1 \cup \mathcal{D}_2, \Delta)}$$

$$\mathit{freeEqn}([\mathtt{u}_i/\mathtt{v}_i]_{i=1}^n, V) \stackrel{\text{def}}{=} \mathtt{let}\ \pi_i = (\mathtt{if}\ \mathtt{v}_i \in V\ \mathtt{then}\ \mathtt{true}\ \mathtt{else}\ \mathtt{v}_i = \mathtt{u}_i)\ \mathtt{in}\ \textstyle\bigwedge_{i=1}^n \pi_i$$

$$\mathit{match}(\mathtt{R}(\mathtt{x}_1, \overline{\Phi_{\mathtt{f}}}^1, \bar{\mathtt{v}}_1), \mathtt{R}(\mathtt{x}_2, \overline{\Phi_{\mathtt{f}}}^2, \bar{\mathtt{v}}_2)) \stackrel{\text{def}}{=} [\mathtt{x}_1/\mathtt{x}_2, \bar{\mathtt{v}}_1/\bar{\mathtt{v}}_2]$$

Fig. 8. Additional Entailment Rules

B A Verified `CountDownLatch` Implementation

With the interpretations for the flow-aware predicates, we can now prove soundness of our normalization, splitting and inconsistency lemmas.

We must also verify the correctness of `CountDownLatch` implementation. A more realistic implementation will make use of locks, wait and notifyAll operations to implement blocking of `await` commands until the shared counter reaches 0. For

ease of presentation, we have used a simple version to illustrate how thread-local abstraction was used, as shown below.

```
CountDownLatch create_latch(int n) {return new CountDownLatch(n); }
void countDown(CountDownLatch i) { ⟨if (i.val>0) i.val = i.val-1;⟩ }
void await(CountDownLatch i) { while (i.val>0) skip; }
```

For ease of presentation, we have used this simple version to illustrate how thread-local abstraction are used. Our verification tool can automatically verify this implementation of `CountDownLatch`. The detailed proof steps are re-produced below.

```
CountDownLatch create_latch(int n) with P
      requires n>0
      ensures LatchIn(res, P)*LatchOut(res, P)*CNT(res,n)
{
    # n>0
    CountDownLatch i = new CountDownLatch(n);
    # i↦CountDownLatch(n)∧n>0
    # [DEC]_1*[KILL]_1*i↦n∧n>0
    # [DEC]_1*{|i↦n|}*i↦n∧n>0
    # [DEC]_1*{|i↦n|}*(P−⊛emp)*P*i↦n∧n>0
    # (P−⊛emp)* [DEC]_1*i↦n*(i↦0 −⟶ P)*{|i↦n|}∧n>0
    # LatchIn(i, P)*LatchOut(i, P)*CNT(i, n)
    return i;
    # LatchIn(res, P)*LatchOut(res, P)*CNT(res, n)
}

CountDownLatch create_latch(int n) with P
      requires n=0
      ensures CNT(res,-1)
{
    # n=0
    CountDownLatch i = new CountDownLatch(n);
    # i↦CountDownLatch(n)∧n=0
    # i↦n∧n=0
    # CNT(i, -1)
    return i;
    # CNT(res, -1)
}
```

```
void countDown(CountDownLatch i)
      requires LatchIn(i,P)*P*CNT(i,n)∧n>0
      ensures CNT(i,n-1);
{
     # LatchIn(i,P)*P*CNT(i,n)∧n>0
     # (P−⊛emp)*[DEC]ϵ* i↦m ∧m>0 * P * {|i↦n|} ∧n>0
     # [DEC]ϵ* i↦m ∧m>0 * {|i↦n|} ∧n>0
     ⟨if (i.val>0) i.val=i.val-1;⟩
     # i↦m-1 ∧m>0 * {|i↦n-1|} ∧n>0
     # {|i↦n-1|} ∧n>0
     # CNT(i,n-1)
}

void countDown(CountDownLatch i)
      requires CNT(i,-1)
      ensures CNT(i,-1);
{
     # CNT(i,-1)
     # i↦0
     ⟨if (i.val>0) i.val=i.val-1;⟩
     # i↦0
     # CNT(i,-1)
}

void await(CountDownLatch i)
      requires LatchOut(i,P)*CNT(i,0)
      ensures P*CNT(i,-1);
{
     # LatchOut(i,P)*CNT(i,0)
     # ( i↦0 −⟶ P)*{|i↦0|}
     # ( i↦0 −⟶ P)* i↦m ∧m≥0
     while (i.val>0) skip;
     # ( i↦0 −⟶ P)* i↦0
     # P*CNT(i,-1)
}


void await(CountDownLatch i)
      requires CNT(i,-1)
      ensures CNT(i,-1);
{
     # CNT(i,-1)
     # i↦0
     while (i.val>0) skip;
     # i↦0
     # CNT(i,-1)
}
```

C Proof of Lemmas for a CDL Library

With the interpretations for the flow-aware predicates, we can now prove soundness of our normalization, splitting and inconsistency lemmas, as follows:

Proof ([SPLIT−1]).

$$\begin{array}{l}
\texttt{LatchOut(i,(P*Q))} \\
\Leftrightarrow \boxed{\texttt{i}\mapsto\texttt{0}} \longrightarrow \texttt{(P*Q)} \\
\Leftrightarrow \boxed{\texttt{i}\mapsto\texttt{0}} \longrightarrow \texttt{P} * \boxed{\texttt{i}\mapsto\texttt{0}} \longrightarrow \texttt{Q} \\
\Leftrightarrow \texttt{LatchOut(i,P)*LatchOut(i,Q)}
\end{array}$$

Proof ([SPLIT−2]).

$$\begin{array}{l}
\texttt{LatchIn(i,(P*Q))} \\
\Leftrightarrow (\texttt{(P*Q)}\text{—}\circledast\texttt{emp})*[\texttt{DEC}]_{\epsilon} * \boxed{\texttt{i}\mapsto\texttt{n}} \wedge \texttt{n>0} \\
\Leftrightarrow (\texttt{P}\text{—}\circledast\texttt{emp})*(\texttt{Q}\text{—}\circledast\texttt{emp})*[\texttt{DEC}]_{\epsilon_1}*[\texttt{DEC}]_{\epsilon_2} * \boxed{\texttt{i}\mapsto\texttt{n}} * \boxed{\texttt{i}\mapsto\texttt{n}} \wedge \texttt{n>0} \wedge \epsilon = \epsilon_1 + \epsilon_2 \\
\Leftrightarrow (\texttt{P}\text{—}\circledast\texttt{emp})*[\texttt{DEC}]_{\epsilon_1} * \boxed{\texttt{i}\mapsto\texttt{n}} \wedge \texttt{n>0} * (\texttt{Q}\text{—}\circledast\texttt{emp})*[\texttt{DEC}]_{\epsilon_2} * \boxed{\texttt{i}\mapsto\texttt{n}} \wedge \texttt{n>0} \\
\Leftrightarrow \texttt{LatchIn(i,P)*LatchIn(i,Q)}
\end{array}$$

Proof ([SPLIT−3]).

$$\begin{array}{l}
\texttt{CNT(i,n)} \wedge \texttt{n1,n2} \geq \texttt{0} \wedge \texttt{n=n1+n2} \\
\Rightarrow \{\!|\texttt{i}\mapsto\texttt{n}|\!\} \wedge \texttt{n1,n2} \geq \texttt{0} \wedge \texttt{n=n1+n2} \\
\Rightarrow \{\!|\texttt{i}\mapsto\texttt{n1}|\!\} * \{\!|\texttt{i}\mapsto\texttt{n2}|\!\} \wedge \texttt{n1,n2} \geq \texttt{0} \\
\Rightarrow \texttt{CNT(i,n1)*CNT(i,n2)}
\end{array}$$

Proof ([NORM−1]).

$$\begin{array}{l}
\texttt{CNT(c,n)*CNT(c,-1)} \wedge \texttt{n} \leq \texttt{0} \\
\Leftrightarrow (\{\!|\texttt{c}\mapsto\texttt{n}|\!\} \wedge \texttt{n} \geq \texttt{0} \vee \boxed{\texttt{c}\mapsto\texttt{0}} \wedge \texttt{n=-1}) * \texttt{CNT(c,-1)} \wedge \texttt{n} \leq \texttt{0} \\
\Leftrightarrow (\{\!|\texttt{c}\mapsto\texttt{n}|\!\} \wedge \texttt{n=0} \vee \boxed{\texttt{c}\mapsto\texttt{0}} \wedge \texttt{n=-1}) * \texttt{CNT(c,-1)} \\
\Rightarrow (\boxed{\texttt{c}\mapsto\texttt{m}} \wedge \texttt{m} \geq \texttt{0} \vee \boxed{\texttt{c}\mapsto\texttt{0}} \wedge \texttt{n=-1}) * \boxed{\texttt{c}\mapsto\texttt{0}} \\
\Rightarrow \boxed{\texttt{c}\mapsto\texttt{0}} \\
\Rightarrow \texttt{CNT(c,-1)}
\end{array}$$

Proof ([NORM−2]).

$$\begin{array}{l}
\texttt{CNT(c,n1)*CNT(c,n2)} \wedge \texttt{n=n1+n2} \wedge \texttt{n1,n2} \geq \texttt{0} \\
\Leftrightarrow \{\!|\texttt{c}\mapsto\texttt{n1}|\!\} * \{\!|\texttt{c}\mapsto\texttt{n2}|\!\} \wedge \texttt{n=n1+n2} \wedge \texttt{n1,n2} \geq \texttt{0} \\
\Leftrightarrow \{\!|\texttt{c}\mapsto\texttt{n}|\!\} \wedge \texttt{n=n1+n2} \wedge \texttt{n1,n2} \geq \texttt{0} \\
\Rightarrow \texttt{CNT(c,n)}
\end{array}$$

Proof ([NORM−3]).

$$\begin{array}{l}
\texttt{LatchOut(c,P)*CNT(c,-1)} \\
\Leftrightarrow (\boxed{\texttt{c}\mapsto\texttt{0}} \longrightarrow \texttt{P}) * \boxed{\texttt{c}\mapsto\texttt{0}} \\
\Rightarrow \texttt{P} * \boxed{\texttt{c}\mapsto\texttt{0}} \\
\Rightarrow \texttt{P*CNT(c,-1)}
\end{array}$$

Proof ($\lceil$**ERR−1**$\rceil$).

$$
\begin{array}{l}
\texttt{LatchIn(c,P)*CNT(c,-1)} \\
\Leftrightarrow \texttt{(P}-\circledast\texttt{emp)*}[\texttt{DEC}]_{\epsilon}\texttt{*}\boxed{\texttt{c}\mapsto\texttt{n}}\texttt{}\wedge\texttt{n>0*CNT(c,-1)} \\
\Leftrightarrow \texttt{(P}-\circledast\texttt{emp)*}[\texttt{DEC}]_{\epsilon}\texttt{*}\boxed{\texttt{c}\mapsto\texttt{n}}\texttt{}\wedge\texttt{n>0*}\boxed{\texttt{c}\mapsto\texttt{0}}\texttt{} \\
\Leftrightarrow \texttt{(P}-\circledast\texttt{emp)*(}[\texttt{DEC}]_{\epsilon}\texttt{*}\boxed{\texttt{c}\mapsto\texttt{n}}\texttt{}\wedge\texttt{n>0*}\boxed{\texttt{c}\mapsto\texttt{0}}\texttt{}\texttt{)} \\
\Leftrightarrow \texttt{(P}-\circledast\texttt{emp)*false} \\
\Leftrightarrow \texttt{false}\quad\texttt{//RACE-ERROR}
\end{array}
$$

Proof ($\lceil$**ERR−2**$\rceil$).

$$
\begin{array}{l}
\texttt{CNT(c,a)*CNT(c,-1)}\wedge\texttt{a>0} \\
\Leftrightarrow \{\lfloor\texttt{c}\mapsto\texttt{a}\rfloor\}\texttt{*CNT(c,-1)}\wedge\texttt{a>0} \\
\Rightarrow \boxed{\texttt{c}\mapsto\texttt{m}}\texttt{}\wedge\texttt{m}\geq\texttt{a*}\boxed{\texttt{c}\mapsto\texttt{0}}\texttt{}\wedge\texttt{a>0} \\
\Rightarrow \texttt{false}\quad\texttt{//DEADLOCK-ERROR}
\end{array}
$$

The cycle detection lemma in $\lceil$**ERR−3**$\rceil$ is a kind of contradiction detection mechanism too, as explained in Sec 3.2.