Verifying the Correctness of CountDownLatch with Flow-Aware Concurrent Predicates

W-N Chin[1] D-K Le[1] T.C. Le[1] S Qin[2] Q-T Ta[1]

[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 synchronize by waiting for some tasks to be completed by other threads. In this paper, we propose a novel framework for verifying the correctness of concurrent applications that use CDLs. Our framework can help ensure that concurrent applications using them are *race-free*, *resource-preserving* and *deadlock-free*. To support this in a general way, we propose a variant of "*flow-aware*" concurrent predicates which can explicitly track resources that flow into and out of shared abstraction boundary for concurrent programs. This resource tracking feature can guarantee race-freedom and resource-preservation. Furthermore, by tracking wait-for conditions between threads via our variant of concurrent predicates, we show how deadlock-freedom can be also guaranteed for such concurrent programs. We have implemented flow-aware predicates in a tool, called ConcHIP, on top of an existing ParaHIP verifier. We have used ConcHIP to verify a suite of sample programs with different usage scenarios for CDLs.

1 Introduction

One of the most popular techniques for reasoning about concurrent programs is separation logic [22, 24]. Originally, separation logic was used to verify heap-manipulating sequential programs, with the ability to express non-aliasing in the heap [24]. Separation logic was also extended to verify shared-memory concurrent programs, e.g. concurrent separation logic [22], 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 [22], locks [10, 11] and first-class threads [18, 9]. Lately, some solutions are beginning to emerge for more complex concurrency mechanisms, such as barriers [13] and channels [7]. 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. As pointed in [27], higher-order concurrent abstract predicates may be unsound (or unstable) if they had been *state-dependent*[1] or uses *self-referential region assertions*[2].

Moreover, we are not aware of any formal verification solution that could handle the more versatile `CountDownLatch` where multiple threads are involved in data exchanges during concurrency synchronization. On its formal correctness, it is important to ensure *race-freedom*, *deadlock-freedom* and *resource-preservation*. Most existing solutions on concurrency verification have focused on only race-freedom, whilst some solutions [19, 17] have been proposed for verifying deadlock-freedom, mostly in the context of mutex locks and channels. We believe it is also important to guarantee resource-preservation to help ensure that concurrent programs do not leak resources and to ensure a sound variant of HOCAP.

In this paper, we propose to use a novel variant of concurrent abstract predicates, called *flow-aware resource* HOCAP, to help model `CountDownLatch`. This solution is based on higher-order separation logic and will use *flow-annotations* and *resource-preservation* to more precisely capture exchanges of resources amongst its concurrent threads. Our proposal for flow-aware resource HOCAP may also be applied to other forms of concurrency mechanisms, including locks, message-passing and barrier synchronization. However, as a focus for the present paper, we show how flow-aware predicates can be used to ensure the correctness of both clients and library implementation for `countDownLatch`. We shall also highlight what concurrency problems are being detected by our approach.

Our paper makes the following technical contributions:

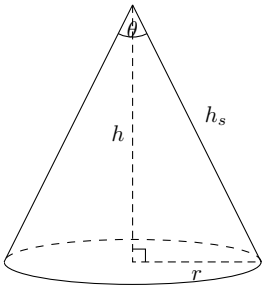
- We propose the first formal verification for `CountDownLatch`. We use a novel variant of concurrent abstract predicate, called *flow-aware* predicates, to help track resources soundly and precisely for our concurrent programs.
- We provide a modular solution to the count down mechanism by supporting a *thread-local* abstraction on top of the usual *global* view on its shared counter.
- We highlight and show three desirable properties (i) race-freedom (ii) resource-preservation and (iii) deadlock-freedom, that can be elicited from verified specifications based on flow-aware resource predicates
- We verified a simple library implementation of the `CountDownLatch`.
- We provide a prototype verifier based on flow-aware concurrent predicates.

[1] An abstract predicate for a protocol is *state-dependent* if it is used in the definition of another abstract predicate.

[2] An assertion used to instantiate a predicate is said to be *self-referential* if it picks instantiation referring to the library's shared region.

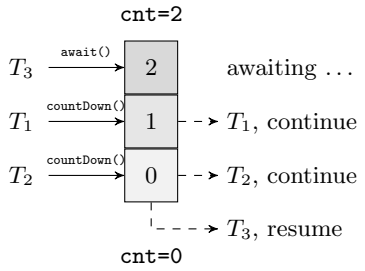
2 Motivation for `CountDownLatch`

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, as shown next.



The first use-case of `CountDownLatch` is when a thread wants to wait for other threads to complete their work, so that all resources needed for its computation are available. As an example, we use `CountDownLatch` to implement a program which calculates the volume V of a right 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 their computations have completed, these threads invoke the `countDown` method to inform the thread T_3, awaiting on the latch, to start its computation for V based on h and r.

```
c = create_latch(2);
(# T1                 || # T2                 || # T3
 # h:=hs cos θ/2;     || # r:=hs sin θ/2;     ||
 countDown(c);       || countDown(c);        || await(c);
                     ||                      || # V:=1/3 π r^2 h;
);
```



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 a simple example, let us assume that resource `P*Q`[3] 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
 ...                || ...                || ...
);
```

[3] `P*Q` is a separation logic formula formed by separation conjunction `*` ([14, 24]).

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);

(  ...                          ||  ...
   # owns P                     ||   # owns Q
   countDown(c); await(c);      ||  countDown(c); await(c);  );
   # owns Q                     ||   # owns P
   ...                          ||  ...
```

As seen with these examples, the communication patterns here are non-trivial and we shall see how flow-aware resource predicates can help us perform both precise and systematic reasoning.

3 Flow-Aware Concurrent Abstract Predicates

In this section, we show how flow-aware predicates can be used to formally model the `CountDownLatch`. Apart from tracking resources more precisely to support race-freedom, deadlock-freedom and resource-preservation, one of the key challenges faced by the `CountDownLatch` concurrency protocol is the ability to support thread modular reasoning for its shared counter. This aspect is one reason why prior approaches to concurrency verification have not yet solved the `CountDownLatch` problem. We introduce three abstract predicates, $\texttt{Latch}(\texttt{c}, \ominus\texttt{P})$, $\texttt{Latch}(\texttt{c}, \oplus\texttt{P})$, $\texttt{CNT}(\texttt{c}, \texttt{n})$, used by the creation of each `CountDownLatch`, as follows:

```
CountDownLatch create_latch(n) with P
      requires n>0
      ensures Latch(res, ⊖P)*Latch(res, ⊕P)*CNT(res,n);
      requires n=0
      ensures CNT(res,-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 $\texttt{CNT}(\texttt{res}, \texttt{-1})$. In case `n>0`, the first two flow-aware resource predicates, $\texttt{Latch}(\texttt{c}, \ominus\texttt{P})$ and $\texttt{Latch}(\texttt{c}, \oplus\texttt{P})$, are used to model the inflow (denoted by $\ominus$) and outflow (denoted by $\oplus$) of some resource `P` that are being exchanged by the `CountDownLatch`. Specifically, the predicate $\texttt{Latch}(\texttt{c}, \ominus\texttt{P})$ shall be used for the *consumption* of resource `P` into the `CountDownLatch` (at `countDown` call), while the predicate $\texttt{Latch}(\texttt{c}, \oplus\texttt{P})$ shall be used to model the *production* of `P` from the `CountDownLatch` (at each `await` call), as shown in the next two methods.

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

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

The resources, denoted by higher-order formula `P`, are logical ones added by our specifications for `CountDownLatch` to support race-free resource exchanges, as the underlying specification of `CountDownLatch` is focused exclusively on its countdown counter and the blocking effects on threads which issue `await` calls. The predicate `CNT(c,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 `CNT(c,n)`$\wedge$`n`$\geq$`0` gives a thread-local view of the counter with a value of at least `n`. As a special case, `CNT(c,0)` denotes a shared counter with at least `0`, while `CNT(c,-1)` denotes a shared counter that is definitely `0` (i.e. the counter has reached its terminal state with value `0`). In order to support resource consumption by the first pre/post specification of `countDown` method, we require `CNT(c,n)`$\wedge$`n`$>$`0` to ensure race-free synchronization.

The following normalization rules show how multiple instances `CNT` 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 `CNT(c,-1)`. The second rule combines multiple `CNT` instances, where possible. The third rule allows resource trapped in each latch to be released at its terminal state.

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

To support multiple concurrent threads, we provide a set of splitting lemmas for our flow-aware predicates, as follows:

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

Our splitting process shall be 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 expected in a thread, our split will try to ensure that a suitable thread-local state, say `Latch(i,`$\oplus$`P)*CNT(i,0)`, be passed to this thread.

We illustrate an example below where splitting lemmas allow relevant abstract predicates to be made available for the modular verification of each thread.

```
                    c = create_latch(2) with P*Q;
                # Latch(c,⊕(P*Q))*Latch(c,⊖(P*Q))*CNT(c,2)
    # Latch(c,⊕(P*Q))*Latch(c,⊖P)*Latch(c,⊖Q)*CNT(c,0)*CNT(c,1)*CNT(c,1)
(  ...                          ||  ...create P...              ||  ...create Q...
 # Latch(c,⊕(P*Q))*CNT(c,0)     ||  # P*Latch(c,⊖P)*CNT(c,1)    ||  # Q*Latch(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...                  ||  ...                         ||  ...                    );
```

Let us now define the syntax of our flow-aware predicates, as follows:

Definition 1 (Syntax of Flow-Aware Predicates) *A* flow-aware *predicate is a concurrent abstract predicate which may carry zero or more flow-annotated resources,* $\overline{\Phi_{\mathtt{f}}}$*. We can denote its syntax by* $\mathtt{R}(\mathtt{x}, \overline{\Phi_{\mathtt{f}}}, \bar{\mathtt{v}})$ *where* $\Phi_{\mathtt{f}}$ *is defined as:*

$\Phi_{\mathtt{f}} ::= \delta\,\mathtt{V} \mid \cdots$

$\delta \quad ::= \odot \mid \ominus \mid \oplus$

Take note that $\overline{\Phi_{\mathtt{f}}}$ may be an empty list, since each concurrent abstract predicate is an instance of our flow-aware resource predicates. The first parameter of every predicate denotes the root pointer to our resource predicate. Note that $\ominus\mathtt{V}$ uses a logical variable $\mathtt{V}$ to denote a resource that is to be transferred into the protocol, while $\oplus\mathtt{V}$ denotes a resource that is to be transferred out of the protocol. We also use annotation $\odot\mathtt{V}$ to capture a resource $\mathtt{V}$ that is to be used as a reference, rather than for resource transfer purposes. More details on the structure of logical formula $\Phi_{\mathtt{f}}$ are described in Sec 4.

3.1 Ensuring Resource Preservation

Concurrency protocols which synchronize resources between multiple threads are expected to be leak-free. We therefore require every lemma and each pre/post specification used be resource-preserving. If this were not the case, then the concurrency protocol either leak resources or are allowed to transform those that pass through its protocols. For example, copy-oriented message-passing channel would create new resource to duplicate each resource that has passed through its send operation. Such protocols may thus have new resources attributed to their pre/post specifications but their effects should be clearly identified.

Other than such protocols, all other concurrency protocols (such as `CountDownLatch`) are expected to be resource-preserving. Our use of flow-annotation is *essential* for ensuring that each concurrency primitive and each lemma used in our reasoning is resource-preserving. To formalize this concept, we introduce the following operator `RS` which is used to compute net resource variables that are captured by a higher-order logic formula:

$$
\begin{array}{llll}
\mathtt{RS}(\Phi_1 * \Phi_2) & \stackrel{\text{def}}{=} \mathtt{RS}(\Phi_1) \cup \mathtt{RS}(\Phi_2) & \mathtt{RS}(\mathtt{R}(\mathtt{x}, [], \bar{\mathtt{v}})) & \stackrel{\text{def}}{=} \{\} \\
\mathtt{RS}(\mathtt{V}) & \stackrel{\text{def}}{=} \{\oplus\mathtt{V}\} & \mathtt{RS}(\Phi) - \{\} & \stackrel{\text{def}}{=} \mathtt{RS}(\Phi) \\
\mathtt{RS}(\mathtt{R}(\mathtt{x}, (\oplus\mathtt{V})::\overline{\Phi_{\mathtt{f}}}, \bar{\mathtt{v}})) & \stackrel{\text{def}}{=} \{\oplus\mathtt{V}\} \cup \mathtt{RS}(\mathtt{R}(\mathtt{x}, \overline{\Phi_{\mathtt{f}}}, \bar{\mathtt{v}})) & \mathtt{RS}(\Phi_1) - (\{\oplus\mathtt{V}\} \cup \mathcal{B}) & \stackrel{\text{def}}{=} \mathtt{RS}(\Phi_1) \cup \{\ominus\mathtt{V}\} - \mathcal{B} \\
\mathtt{RS}(\mathtt{R}(\mathtt{x}, (\ominus\mathtt{V})::\overline{\Phi_{\mathtt{f}}}, \bar{\mathtt{v}})) & \stackrel{\text{def}}{=} \{\ominus\mathtt{V}\} \cup \mathtt{RS}(\mathtt{R}(\mathtt{x}, \overline{\Phi_{\mathtt{f}}}, \bar{\mathtt{v}})) & \mathtt{RS}(\Phi_1) - (\{\ominus\mathtt{V}\} \cup \mathcal{B}) & \stackrel{\text{def}}{=} \mathtt{RS}(\Phi_1) \cup \{\oplus\mathtt{V}\} - \mathcal{B} \\
\mathtt{RS}(\mathtt{R}(\mathtt{x}, (\_)::\overline{\Phi_{\mathtt{f}}}, \bar{\mathtt{v}})) & \stackrel{\text{def}}{=} \mathtt{RS}(\mathtt{R}(\mathtt{x}, \overline{\Phi_{\mathtt{f}}}, \bar{\mathtt{v}})) & \{\ominus\mathtt{V}, \oplus\mathtt{V}\} \cup \mathcal{B} & \Rightarrow \mathcal{B} \quad \text{(simplification)}
\end{array}
$$

A pre/post condition of form `requires` Φ_1 `ensures` Φ_2 is said to be *resource-preserving* if $\mathtt{RS}(\Phi_2) - \mathtt{RS}(\Phi_1) = \{\}$. As an example, the specification of `await` is resource-preserving since $\mathtt{RS}(\mathtt{CNT(c,-1)} * \mathtt{P}) - \mathtt{RS}(\mathtt{Latch(c,}\oplus\mathtt{P)} * \mathtt{CNT(c,0)}) = \{\oplus\mathtt{P}, \ominus\mathtt{P}\} = \{\}$. Similarly, each lemma $\Phi_1 \longrightarrow \Phi_2$ is said to be resource-preserving if $\mathtt{RS}(\Phi_2) - \mathtt{RS}(\Phi_1) = \{\}$. All normalization and splitting lemmas used for

our CountDownLatch problem can be shown to be resource-preserving. In summary, once a concurrency library specification has been shown to be resource-preserving, it is straightforward to ensure that application codes are also resource-preserving through the use of classical separation logic. This helps guarantee that concurrency programs are leak-free, which is a critical property for languages that do not use automatic garbage collection, such as C.

3.2 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 $\texttt{CNT}(\texttt{c},\texttt{n})\wedge\texttt{n}>0$ or $\texttt{CNT}(\texttt{c},\texttt{-1})$, but never $\texttt{CNT}(\texttt{c},\texttt{0})$. Thus, each violation on the pre-condition of countDown[4] would be signaled as a potential race problem. There is another way that race problem can occur when resources that are required by await calls are not being synchronized by a countDown call. An example to illustrate this is shown below.

```
                    c = create_latch(1) with P*Q;
                # Latch(c,⊕(P*Q))*Latch(c,⊖(P*Q))*CNT(c,1)
# Latch(c,⊕(P*Q))*Latch(c,⊖P)*Latch(c,⊖Q)*CNT(c,0)*CNT(c,1)*CNT(c,0)

(  ···                          ||  ···create P···              ||  ···create Q···
# Latch(c,⊕(P*Q))*CNT(c,0)      ||# P*Latch(c,⊖P)*CNT(c,1)      ||# Q*Latch(c,⊖Q)*CNT(c,0)
   await(c);                    ||  countDown(c);               ||  skip();
# P*Q*CNT(c,-1)                 ||# CNT(c,0)                    ||# Q*Latch(c,⊖Q)*CNT(c,0)
   ···use P*Q···                ||  ···                         ||  ···                        );

          # P*Q*CNT(c,-1) *CNT(c,0) *Q*Latch(c,⊖Q)*CNT(c,0)
                    # P*Q*CNT(c,-1) *Q*Latch(c,⊖Q)
                  # RACE-ERROR detected by [ERR−1]
```

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

$[\textbf{ERR}{-}1]$: $\texttt{Latch}(\texttt{c},\ominus\texttt{P})*\texttt{CNT}(\texttt{c},\texttt{-1}) \longrightarrow \texttt{RACE-ERROR}$

The formula $\texttt{Latch}(\texttt{c},\ominus\texttt{P})*\texttt{CNT}(\texttt{c},\texttt{-1})$ denotes a contradiction. The former predicate requires the shared counter to be non-zero, while latter predicate is a terminal state with value 0. Such contradictions are manifestations of some concurrency synchronization errors. We identify this as a race problem.

[4] This violation occurs when skip() in the third thread is replaced by countDown(c).

3.3 Ensuring Deadlock Freedom

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

Within a single `CountDownLatch`, this deadlock error can be signified by:

$[\textbf{ERR}-\textbf{2}]$: $\texttt{CNT(c,a)*CNT(c,-1)}\wedge\texttt{a>0} \longrightarrow \texttt{DEADLOCK-ERROR}$

Here, $\texttt{CNT(c,a)}\wedge\texttt{a>0}$ denotes a counter value of at least 1, while $\texttt{CNT(c,}-\texttt{1)}$ denotes a final counter with value 0. Such a contradiction is a manifestation of a deadlock error that arose from the unreachability on precondition $\texttt{CNT(c,a)}\wedge\texttt{a}\leq\texttt{0}$ that we impose on the `await` method.

A simple example of this deadlock scenario (omitting `Latch` predicates) is:

```
        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]
```

This deadlock error occurs due to the first thread not invoking sufficient number of `countDown` calls. This lack of `countDown` calls led to an error scenario when the conflicting states of the two threads are joined.

For multiple latches, we will need to track a wait-for graph [25], to help detect deadlocks, where possible. Let us introduce a `WAIT` relation that is tracked inter-procedurally with its permission. Whenever we have a complete view of the waits-for relations (with full permission), we can always reset each acyclic wait-for graph to {}, as follows:

$[\textbf{WAIT}-\textbf{1}]$: $\texttt{WAIT(S)}_1\wedge\neg\texttt{isCyclic(S)}\longrightarrow \texttt{WAIT}(\{\})_1$

We add a wait-for arc via this lemma:

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

$\{c_2\rightarrow c_1\}$ 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 splitted/normalized by:

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

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 lemma:

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

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

```
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]
```

Though deadlock detection for locks and channels have been proposed before [19], there are at least two novel ideas in our present proposal. Firstly, we have now shown how to formally ensure deadlock-freedom for the complex `CountDownLatch` protocol. Secondly, we have achieved this through the use of contradiction lemmas. The wait-for arcs added are used for contradiction detection, since each arc $c_2 \rightarrow c_1$ denotes a strict completion ordering $c_2 < c_1$ (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.

3.4 A Thread-Local Abstraction for `CountDownLatch`

We have introduced three concurrent abstract predicates for the specification of `CountDownLatch`. Two of them carry flow-aware resources, while the third one is used to track its shared counter. A simple implementation of this `CountDownLatch` concurrency protocol can be described by the following methods.

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

Of the two concurrent methods, only the `countDown` method may cause interference. Following [6], this potential interference can be described by a `DEC` action where the shared global counter $\boxed{\texttt{i}\mapsto\texttt{n}}$ is decreased by `1` if it is non-zero.

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

We use this implementation to show how the flow-aware resource predicates may be defined, and also to illustrate how the correctness of `CountDownLatch` may be verified. In the `countDown` method, this action is currently wrapped by an atomic operation, marked with $\langle\cdots\rangle$. While this `DEC` action is formulated using the global view of shared counter, it would be better to support a *thread-local* view for such shared counters. For this, we introduce a new *thread-local* abstraction $\{\!|\texttt{i}\mapsto\texttt{n}|\!\}$ to denote a shared counter that is *always* $\texttt{n}{\geq}0$. Such a thread-local abstraction is related to its global counter by the following property.

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

As this is a thread-local abstraction, we shall also provide the following combine/split operations that can be used to normalize or split thread-local counters.

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

Contrast this to global view of shared counter with the conjunctive property:
$\boxed{\mathtt{i}\mapsto\mathtt{a}}*\boxed{\mathtt{i}\mapsto\mathtt{b}} \longleftrightarrow \boxed{\mathtt{i}\mapsto\mathtt{a}}\wedge\mathtt{a=b}$

With this use of thread-local abstraction, we are now ready to provide interpretations for our flow-aware predicates that are stable, as follows:

$\mathtt{Latch}(\mathtt{i},\ominus\mathtt{P}) \stackrel{\text{def}}{=} [\mathtt{DEC}]_\epsilon*\ominus\mathtt{P}*\boxed{\mathtt{i}\mapsto\mathtt{n}}\wedge\mathtt{n>0}$

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

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

We can always attach a fractional permission ϵ to the DEC action, to signify that the current thread has permission ϵ to perform the action. For ease in presentation, we omit its tracking at the predicate level. Our use of $\{|\mathtt{i}\mapsto\mathtt{n}|\}$ remains stable since it is based on thread-local view. The shared global state $\boxed{\mathtt{i}\mapsto\mathtt{0}}$ is also stable, as the DEC operation does not modify this final state of our shared counter. The shared $\boxed{\mathtt{i}\mapsto\mathtt{m}}\wedge\mathtt{m>0}$ in $\mathtt{Latch}(\mathtt{i},\ominus\mathtt{P})$ is stable since it is always used with $\mathtt{CNT}(\mathtt{i},\mathtt{n})$ in the pre-condition of countDown method. This results in $\boxed{\mathtt{i}\mapsto\mathtt{m}}\wedge\mathtt{m>0}*\{|\mathtt{i}\mapsto\mathtt{n}|\}\wedge\mathtt{n>0}$ which is stable, despite interference from other $[\mathtt{DEC}]_\epsilon$. $\mathtt{Latch}(\mathtt{i},\oplus\mathtt{P})$ is stable since it depends on a pure and stable $\boxed{\mathtt{i}\mapsto\mathtt{0}}$.

For the purpose of creation and later destroying this shared global counter, we provided another action, called KILL action, that allows a global counter to be made local by the following equivalence lemma:

$$\begin{aligned}\mathtt{i}\mapsto\mathtt{CountDownLatch(n)}\wedge\mathtt{n}\geq\mathtt{0} &\longleftrightarrow [\mathtt{DEC}]_1*[\mathtt{KILL}]_1*\boxed{\mathtt{i}\mapsto\mathtt{n}}\wedge\mathtt{n}\geq\mathtt{0}\\ &\longleftrightarrow [\mathtt{DEC}]_1*\{|\mathtt{i}\mapsto\mathtt{n}|\}_1*\boxed{\mathtt{i}\mapsto\mathtt{n}}\end{aligned}$$

This KILL action permits the creation of a thread-local heap state. By tracking the permissions of both DEC action and thread-local counter, we could recover the full permission needed for the thread-local counter to be converted to just the global view, prior to its conversion back to a local heap for explicit disposal.

3.5 A Verified CountDownLatch Implementation

With the interpretations for the flow-aware predicates, we can now prove soundness of our normalization, splitting and contradiction lemmas. We give the proof for [SPLIT−2] here, with the rest in Appendix A.

Proof ([SPLIT−2]).

$\mathtt{Latch}(\mathtt{i},\ominus(\mathtt{P*Q}))$

$\Leftrightarrow [\mathtt{DEC}]_\epsilon*\ominus(\mathtt{P*Q})*\boxed{\mathtt{i}\mapsto\mathtt{n}}\wedge\mathtt{n>0}$

$\Rightarrow [\mathtt{DEC}]_{\epsilon_1}*[\mathtt{DEC}]_{\epsilon_2}*\ominus\mathtt{P}*\ominus\mathtt{Q}*\boxed{\mathtt{i}\mapsto\mathtt{n}}*\boxed{\mathtt{i}\mapsto\mathtt{n}}\wedge\mathtt{n>0}\wedge\epsilon=\epsilon_1+\epsilon_2$

$\Rightarrow \mathtt{Latch}(\mathtt{i},\ominus\mathtt{P})*\mathtt{Latch}(\mathtt{i},\ominus\mathtt{Q})$

We must also verify the correctness of CountDownLatch implementation. A more realistic implementation will make use of locks, wait and notifyAll operation 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 were used. We highlight the verification steps for one pre/post specification of countDown. The rest are left in Appendix C.

```
void countDown(CountDownLatch i)
      requires Latch(i, ⊖P)*P*CNT(i,n)∧n>0
      ensures CNT(i,n-1);
{
     # Latch(i, ⊖P)*P*CNT(i,n)∧n>0
     # [DEC]ϵ* i↦m ∧m>0* ⊖ P *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)
}
```

4 Formalism of Language and Logic

Program	*Prog*	::=	$\overline{datat}$ $\overline{proc}$
Data declaration	*datat*	::=	**data** C { $\overline{t\ f}$ }
Procedure declaration	*proc*	::=	$t\ pn(\overline{t\ v})$ $\overline{spec}$ { s }
Pre/Post-conditions	*spec*	::=	requires Φ_{pr} ensures Φ_{po};
Type	t	::=	void | int | bool | CountDownLatch | C
Expression	e	::=	$v \mid v.f \mid k \mid$ new C($\overline{v}$) $\mid e_1; e_2 \mid e_1 \| e_2 \mid \langle e \rangle$
			create_latch(n) with $\kappa \wedge \pi$ | countDown(v) |
			await(v) | $pn(\overline{v})$ | if v then e_1 else e_2 | ...

Fig. 1. Core Language with CountDownLatch

We use the core language in Fig. 1 to formalise our reasoning with flow-aware resource predicates. In this paper, we restrict our attention to CountDownLatch. A program consists of data declarations ($\overline{datat}$), and procedure declarations ($\overline{proc}$). 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.

Fig. 2 shows our specification language for concurrent programs supporting "flow-aware" 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

$$
\begin{array}{rrl}
\text{FA Pred.} & \mathit{rpred} & ::= \texttt{pred R(self},\bar{\texttt{V}},\bar{\texttt{v}}) \ [\equiv\ \Phi]\ [\textbf{inv}\ \pi] \\
\text{Action} & \mathit{act} & ::= \texttt{action I} \ \equiv\ \overline{\iota_1 \wedge \pi_1 \rightsquigarrow \iota_2 \wedge \pi_2} \\
\text{Disj. formula} & \Phi & ::= \bigvee(\exists \bar{\texttt{v}} \cdot \eta * \kappa \wedge \pi) \\
\text{NonFlow formula} & \eta & ::= [\texttt{I}]_\xi \mid \texttt{WAIT}\ \{\overline{v_1 \rightarrow v_2}\}_\xi \mid \boxed{\texttt{v} \mapsto \texttt{C(}\bar{\texttt{v}}\texttt{)}} \mid \eta_1 * \eta_2 \\
\text{Sep. formula} & \kappa & ::= \iota \mid [\ominus]\,\texttt{V} \mid \texttt{R}(\texttt{v}, \overline{\Phi_{\texttt{f}}}, \bar{\texttt{v}}) \mid \kappa_1 * \kappa_2 \\
\text{Simple heap} & \iota & ::= \texttt{emp} \mid \texttt{v} \mapsto \texttt{C}(\bar{\texttt{v}}) \mid \{\!|\texttt{v} \mapsto \texttt{C}(\bar{\texttt{v}})|\!\} \mid \iota_1 * \iota_2 \\
\text{Flow formula} & \Phi_{\texttt{f}} & ::= \delta\,\texttt{V} \mid \delta\,\kappa \wedge \pi \\
\text{Flow. ann.} & \delta & ::= \odot \mid \ominus \mid \oplus \\
\text{Perms} & \xi & ::= \epsilon \mid 1 \\
\text{Pure formula} & \pi & ::= \alpha \mid \pi_1 \wedge \pi_2 \mid \pi_1 \vee \pi_2 \mid \neg\pi \mid \exists \texttt{v} \cdot \pi \mid \forall \texttt{v} \cdot \pi \\
\text{Arith. formula} & \alpha & ::= \alpha_1^t = \alpha_2^t \mid \alpha_1^t \neq \alpha_2^t \mid \alpha_1^t < \alpha_2^t \mid \alpha_1^t \leq \alpha_2^t \\
\text{Arith. term} & \alpha^t & ::= \texttt{k} \mid \texttt{v} \mid \texttt{k} \times \alpha^t \mid \alpha_1^t + \alpha_2^t \mid -\alpha^t
\end{array}
$$

$k \in$ *integer constants* $\quad \texttt{v} \in$ *variables*, $\bar{\texttt{v}} \equiv \texttt{v}_1, .., \texttt{v}_n \quad \texttt{C} \in$ *data names*

$\texttt{V} \in$ *resource variables* $\quad \texttt{R} \in$ *resource pred. names* $\quad \epsilon \in (0, 1]$

Fig. 2. Core Specification Language

non-flow formula η may comprise action permission assertions $[\texttt{I}]_\xi$, (permission annotated) `WAIT` relations, or global views on shared states $\boxed{\texttt{v} \mapsto \texttt{C(}\bar{\texttt{v}}\texttt{)}}$. The heap formula κ can be formed by simple heaps ι, (flow-annotated) resource variables $[\delta]\,\texttt{V}$, resource predicate instances $\texttt{R}(\texttt{x}, \overline{\Phi_{\texttt{f}}}, \bar{\texttt{v}})$, or via separation conjunction $*$ ([14, 24]). Note that a resource variable `V` is a place holder for a flow formula $\Phi_{\texttt{f}}$, used in a resource predicate declaration, while a resource predicate instance $\texttt{R}(\texttt{x}, \overline{\Phi_{\texttt{f}}}, \bar{\texttt{v}})$ encapsulates in-flow and/or out-flow resources ($\overline{\Phi_{\texttt{f}}}$) that are accessible via `x`. For precision reasons, these flow-aware resources are restricted to local entities that can be tracked precisely, such as heap nodes or abstract predicates, but must not include shared global shared locations or `WAIT` relations. A simple heap ι is formed by data nodes $\texttt{v} \mapsto \texttt{C}(\bar{\texttt{v}})$ (which can also appear in a thread-local view $\{\!|\texttt{v} \mapsto \texttt{C(}\bar{\texttt{v}}\texttt{)}|\!\}$).

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. Initially, the entailment is invoked with $E = \emptyset$, in which case we abbreviate the entailment as $\Delta_A \vdash \Delta_C \rightsquigarrow (\mathcal{D}, \Delta_R)$. E will be built up during the entailment.

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 D. These other rules are adapted from prior works [21, 20] 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.

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

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

$$\frac{[\mathbf{RP{-}IN}]}{\Phi_2 \vdash_E \Phi_1 \rightsquigarrow \mathcal{D}}{\Phi_1 \vdash^{\ominus}_E \Phi_2 \rightsquigarrow \mathcal{D}} \qquad \frac{[\mathbf{RP{-}OUT}]}{\Phi_1 \vdash_E \Phi_2 \rightsquigarrow \mathcal{D}}{\Phi_1 \vdash^{\oplus}_E \Phi_2 \rightsquigarrow \mathcal{D}} \qquad \frac{[\mathbf{RP{-}L{-}INST}]}{\mathtt{V} \in E}{\mathtt{V} \vdash_E \Phi \rightsquigarrow \{(\mathtt{V}, \Phi)\}} \qquad \frac{[\mathbf{RP{-}R{-}INST}]}{\mathtt{V} \in E}{\Phi \vdash_E \mathtt{V} \rightsquigarrow \{(\mathtt{V}, \Phi)\}}$$

In the above 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 $\delta(\rho(\Phi_2))$) 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 flow-directed entailment $\Phi_1 \vdash^{\delta}_{E\cup\{\mathtt{V}\}} \Phi_3 \rightsquigarrow \mathcal{D}$ 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 $\mathtt{V}$ 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.

Rule [**RP−IN**] ensures that the inflow resource is contravariant, while [**RP−OUT**] ensures that the outflow resource is covariant. This directly follows from resource flow principles: A resource P flows into another resource Q if P is more precise than (or entails) Q. This use of flow-annotation to determine the order of entailment is *critical* for ensuring accurate modeling of the resource flows and exchanges. Our flow-directed entailment discovers resource bindings (or instantiation) via [**RP−L−INST**] and [**RP−R−INST**]. This entailment is based on classic entailment without any frame residue. An example of the combination of matching ([**RP−MATCH**]), followed resource binding ([**RP−L−INST**]) is:

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

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

$\texttt{Latch}(\texttt{c}, \oplus(\texttt{x} \mapsto \texttt{cell}(v_1) \ \texttt{*}\ \texttt{y} \mapsto \texttt{cell}(v_2))) \vdash \texttt{Latch}(\texttt{c}, \oplus(\texttt{x} \mapsto \texttt{cell}(v_1)))$
$\rightsquigarrow (\emptyset, \texttt{Latch}(\texttt{c}, \oplus(\texttt{y} \mapsto \texttt{cell}(v_2))))$

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 D.

6 Prototype Implementation

We demonstrate the feasibility of our approach in supporting `CountDownLatch` via flow-aware resource predicates by implementing it on top of ParaHIP[17], a recent verifier for concurrent programs that is able to verify functional correctness (of heap-based programs) and deadlock-freedom. ParaHIP models threads and coarse-grain locks, but do not support flow-aware concurrent predicates, or the more sophisticated concurrency primitives, such as `CountDownLatch`. In contrast, besides verifying functional correctness, data-race freedom, deadlock freedom, our new implementation (called ConcHIP) is now capable of also verifying `CountDownLatch` and beyond through its support for flow-aware concurrent predicates. Our ConcHIP prototype implementation and a set of verified programs (involving `CountDownLatch` and other concurrency mechanisms) are available for both online use and download at:

`http://loris-7.ddns.comp.nus.edu.sg/~project/conchip`

7 Related Work and Conclusion

Traditional works on concurrency verification such as Owicki-Gries [23] and Rely/Guarantee reasoning [16] are focused on using controlled interference to formally reason between each process against those that execute in the background. Subsequent work on concurrency reasoning [22] 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, 15, 19], and use reasoning machineries, such as RGSep [28], 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, resource-preservation and deadlock-freedom.

Our flow-aware resource predicates are based on Concurrent Abstract Predicates (CAP) [6, 7, 27, 26]. 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. Despite being powerful, their specifications may render the reasoning to be unsound in certain corner cases. Svendsen et al. [27] presented a logic called Higher Order Concurrent Abstract Predicates (HOCAP), allowing clients to refine the generic specifications of concurrent data structures. HOCAP was developed based on Jacobs and Piessens' idea of parameterizing specifications of concurrent methods with ghost code, to be executed in synchrnonization points [15]. But instead of resorting to lock invariants from clients as in [15], HOCAP uses higher order (predicative) protocols to allow clients to transfer ownership of additional resources to shared data structures. More recently, Svendsen and Birkedal [26] proposed a more general logic called impredicative Concurrent Abstract Predicates (iCAP) that allows the use of impredicative protocols parameterised on arbitrary predicates and supports modular reasoning about layered and recursive abstractions. Compared with these general frameworks, our flow-aware 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 also proposed a thread-local abstraction which makes it much simpler to ensure stability of our flow-aware concurrent predicates, We have also used contradiction lemma support to ensure deadlock freedom and race-freedom, and have used the flow annotation to 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 [19, 17]. In comparison, we have provided a new approach based on contradiction lemma and wait-for set to ensure deadlock freedom.

In conclusion, we have proposed a framework to guarantee the correctness of concurrent programs using `CountDownLatch`. We showed how to ensure *race-*

freedom, *resource-preservation* 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 introduced *flow-aware predicates* to precisely track resources that are shared by such concurrency primitives. 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 contradiction 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 a new thread-local abstraction for shared counters.

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. S. S. Ishtiaq and P. W. O'Hearn. BI as an Assertion Language for Mutable Data Structures. In *POPL*, 2001.
15. B. Jacobs and F. Piessens. Expressive Modular Fine-grained Concurrency Specification. In *POPL*, 2011.
16. C. B. Jones. Specification and Design of (Parallel) Programs. In *IFIP Congress*, 1983.
17. D.-K. Le, W.-N. Chin, and Y. M. Teo. An Expressive Framework for Verifying Deadlock Freedom. In *ATVA*, 2013.
18. K. R. M. Leino and P. Müller. A Basis for Verifying Multi-threaded Programs. In *ESOP*, 2009.
19. K. R. M. Leino, P. Müller, and J. Smans. Deadlock-Free Channels and Locks. In *ESOP*, 2010.
20. H. H. Nguyen and W.-N. Chin. Enhancing program verification with lemmas. In *CAV*, 2008.
21. H. H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated Verification of Shape and Size Properties via Separation Logic. In *VMCAI*, 2007.
22. P. W. O'Hearn. Resources, Concurrency and Local Reasoning. In *CONCUR*, 2004.
23. S. S. Owicki and D. Gries. Verifying Properties of Parallel Programs: an Axiomatic Approach. *CACM*, 19(5), 1976.
24. J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*, 2002.
25. A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating system concepts*. Wiley, 2013.
26. K. Svendsen and L. Birkedal. Impredicative concurrent abstract predicates. In *ESOP*, 2014.

27. K. Svendsen, L. Birkedal, and M. J. Parkinson. Modular Reasoning about Separation of Concurrent Data Structures. In *ESOP*, 2013.
28. V. Vafeiadis and M. J. Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR*, 2007.

A 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 contradiction lemmas, as follows:

Proof ([SPLIT−1]).
$\texttt{Latch}(\texttt{i}, \oplus(\texttt{P*Q}))$
$\Leftrightarrow \texttt{P*Q}$
$\Leftrightarrow \texttt{Latch}(\texttt{i}, \oplus\texttt{P})*\texttt{Latch}(\texttt{i}, \oplus\texttt{Q})$

Proof ([SPLIT−2]).
$\texttt{Latch}(\texttt{i}, \ominus(\texttt{P*Q}))$
$\Leftrightarrow [\texttt{DEC}]_{\epsilon}*\ominus(\texttt{P*Q})*\boxed{\texttt{i}\mapsto\texttt{n}}\wedge\texttt{n>0}$
$\Rightarrow [\texttt{DEC}]_{\epsilon_1}*[\texttt{DEC}]_{\epsilon_2}*\ominus\texttt{P}*\ominus\texttt{Q}*\boxed{\texttt{i}\mapsto\texttt{n}}*\boxed{\texttt{i}\mapsto\texttt{n}}\wedge\texttt{n>0}\wedge\epsilon=\epsilon_1+\epsilon_2$
$\Rightarrow \texttt{Latch}(\texttt{i}, \ominus\texttt{P})*\texttt{Latch}(\texttt{i}, \ominus\texttt{Q})$

Proof ([SPLIT−3]).
$\texttt{CNT}(\texttt{i},\texttt{n})\wedge\texttt{n1,n2}\geq\texttt{0}\wedge\texttt{n=n1+n2}$
$\Rightarrow \{|\texttt{i}\mapsto\texttt{n}|\}\wedge\texttt{n1,n2}\geq\texttt{0}\wedge\texttt{n=n1+n2}$
$\Rightarrow \{|\texttt{i}\mapsto\texttt{n1}|\}*\{|\texttt{i}\mapsto\texttt{n2}|\}\wedge\texttt{n1,n2}\geq\texttt{0}$
$\Rightarrow \texttt{CNT}(\texttt{i},\texttt{n1})\wedge\texttt{CNT}(\texttt{i},\texttt{n2})$

Proof ([NORM−1]).
$\texttt{CNT}(\texttt{c},\texttt{n})*\texttt{CNT}(\texttt{c},\texttt{-1})\wedge\texttt{n}\leq\texttt{0}$
$\Leftrightarrow (\{|\texttt{c}\mapsto\texttt{n}|\}\wedge\texttt{n}\geq\texttt{0} \vee \boxed{\texttt{c}\mapsto\texttt{0}}\wedge\texttt{n=-1})*\texttt{CNT}(\texttt{c},\texttt{-1})\wedge\texttt{n}\leq\texttt{0}$
$\Leftrightarrow (\{|\texttt{c}\mapsto\texttt{n}|\}\wedge\texttt{n=0} \vee \boxed{\texttt{c}\mapsto\texttt{0}}\wedge\texttt{n=-1})*\texttt{CNT}(\texttt{c},\texttt{-1})$
$\Rightarrow (\boxed{\texttt{c}\mapsto\texttt{m}}\wedge\texttt{m}\geq\texttt{0} \vee \boxed{\texttt{c}\mapsto\texttt{0}}\wedge\texttt{n=-1})*\boxed{\texttt{c}\mapsto\texttt{0}}$
$\Rightarrow \boxed{\texttt{c}\mapsto\texttt{0}}$
$\Rightarrow \texttt{CNT}(\texttt{c},\texttt{-1})$

Proof ([NORM−2]).
$\texttt{CNT}(\texttt{c},\texttt{n1})*\texttt{CNT}(\texttt{c},\texttt{n2}) \wedge \texttt{n=n1+n2} \wedge \texttt{n1,n2}\geq\texttt{0}$
$\Leftrightarrow \{|\texttt{c}\mapsto\texttt{n1}|\}*\{|\texttt{c}\mapsto\texttt{n2}|\} \wedge \texttt{n=n1+n2} \wedge \texttt{n1,n2}\geq\texttt{0}$
$\Leftrightarrow \{|\texttt{c}\mapsto\texttt{n}|\} \wedge \texttt{n=n1+n2} \wedge \texttt{n1,n2}\geq\texttt{0}$
$\Rightarrow \texttt{CNT}(\texttt{c},\texttt{n})$

Proof ([NORM−3]).
$\texttt{Latch}(\texttt{c}, \oplus\texttt{P})*\texttt{CNT}(\texttt{c},\texttt{-1})$
$\Leftrightarrow (\boxed{\texttt{c}\mapsto\texttt{0}} \longrightarrow \texttt{P})*\boxed{\texttt{c}\mapsto\texttt{0}}$
$\Rightarrow \texttt{P}*\boxed{\texttt{c}\mapsto\texttt{0}}$
$\Rightarrow \texttt{P}*\texttt{CNT}(\texttt{c},\texttt{-1})$

Proof ([ERR−1]).
Latch(c, ⊖P)*CNT(c,-1)
⇔ [DEC]$_\epsilon$*⊖P*$\boxed{c\mapsto n}$∧n>0*CNT(c,-1)
⇒ $\boxed{c\mapsto n}$∧n>0*$\boxed{c\mapsto 0}$
⇒ false //RACE-ERROR

Proof ([ERR−2]).
CNT(c,a)*CNT(c,-1)∧a>0
⇒ {|c↦a|}*CNT(c,-1)∧a>0
⇒ $\boxed{c\mapsto m}$∧m≥a*$\boxed{c\mapsto 0}$∧a>0
⇒ false //DEADLOCK-ERROR

The cycle detection lemma in [ERR−3] is a kind of contradiction detection mechanism too, as explained in Sec 3.3.

B Verification Proofs for a CDL Library

We can also verify the correctness of a CountDownLatch implementation. For ease of presentation, we have used a simple version to illustrate how thread-local abstraction were being used. Our verification tool can currently automatically verify this simpler implementation of CountDownLatch. The detailed proof steps are re-produced below.

```
CountDownLatch create_latch(int n) with P
      requires n>0
      ensures Latch(res, ⊖P)*Latch(res, ⊕P)*CNT(res,n)
{
    # n>0
    CountDownLatch i = new CountDownLatch(n);
    # i↦CountDownLatch(n)∧n>0
    # [DEC]1*[KILL]1* ⊖ P* ⊕ P*[i↦n]∧n>0
    # [DEC]1* ⊖ P*([i↦0] ⟶ ⊕ P)*[i↦n]*{|i↦n|}∧n>0
    # Latch(i, ⊖P)*Latch(i, ⊕P)*CNT(i, n)
    return i;
    # Latch(res, ⊖P)*Latch(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 Latch(i,⊖P)*P*CNT(i,n)∧n>0
      ensures CNT(i,n-1);
{
    # Latch(i,⊖P)*P*CNT(i,n)∧n>0
    # [DEC]ϵ* i↦m ∧m>0* ⊖ P *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 Latch(i,⊕P)*CNT(i,0)
      ensures P*CNT(i,-1);
{
    # Latch(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 A CDL Library with Permissions and Recovery

```
   Latch(x,+P)  == ([x->0] -> P)
   Latch{f}(x,-P)  == DEC_f * -P * (x->m) & m>0no
   CNT{f,f2}(x,P) == DEC_f * {|x->n|}_f2 & m>=0
                  \/ DEF_f * {|x->0|}_f2 * [i->0] & n=-1

Question : Do we need to track permission of thread-local resource?
      {|i->n|}


  CDL newCDL(n) with P
   requires n>0
   ensures lc(res,+P) * lc{1}(res,-P) * CNT{1,1}(res,n)
   requires n=0
   ensures  CNT{1,1}(res,-1)

 void countDown(CDL i)
   requires lc{f1}(i,-P)*P*CNT{f2,f3}(i,n) & n>0
   ensures  CNT{f1+f2,f3}(i,n-1);
   requires CNT{f,f2}(i,-1)
   ensures  CNT{f,f2}(i,-1);

  void await(i)
   requires lc(i,+P) * CNT{f,f2}(i,0)
   ensures  P * CNT{f,f2}(i,-1);
   requires CNT{f,f2}(i,-1)
   ensures  CNT{f,f2}(i,-1);

 shared2local(CDL c)
  requires CNT{1,1}(c,-1)
  ensures  c->CDL(0);
{
  # CNT{1,1}(c,-1)
  # DEC_1*[c->0]*{|c->0|}_1
  # DEC_1*KILL_1*[c->0]
  # c->CDL(0)
}


Splitting Lemma:
  lc(i,+P*Q) --> lc(i,+P)*lc(i,+Q)
  lc{f1+f2}(i,-P*Q) --> lc{f1}(i,-P)*lc{f2}(i,-Q)
  CNT{f1+f2,f3+f4}(i,n) & n1,n2>=0 & n=n1+n2
      --> CNT{f1,f3}(i,n1)*CNT{f2,f4}(i,n2)
```

```
Normalization:
  CNT{f1,f2}(c,n) * CNT{f3,f4}(c,-1) & n<=0
    ==> CNT{f1+f3,f2+f4a}(c,-1) // idempotent lemma
  CNT{f1,f2}(c,n1) * CNT{f3,f4}(c,n2) & n=n1+n2
    & n1,n2>=0 & n>=0  ==> CNT{f1+f3,f2+f4}(c,n)
  Latch(c,P) * CNT{f3,f4}(c,-1)
    ==> %P * CNT{f3,f4}(c,-1)

Deadlock detection:
  CNT{_,_}(c,n) * CNT{_,_}(c,-1) & n>0  ==> DEADLOCK
  Latch{_}(c,-P) * CNT(c,-1)  ==> RACE-PROBLEM
```

D Additional Entailment Rules

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

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

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

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

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

Fig. 3. Additional Entailment Rules