Abstract Disjointness and Abstract Atomicity

Pedro da Rocha Pinto[1], Thomas Dinsdale-Young[2], and Philippa Gardner[1]

[1] Imperial College London
{pmd09,pg}@doc.ic.ac.uk
[2] Aarhus University
tyoung@cs.au.dk

Abstract. We look at approaches to specifying concurrent program modules based on *disjointness* (the abstraction that concurrent operations act on disjoint resources) and *atomicity* (the abstraction that concurrent operations occur at disjoint times). We illustrate the limitations of these approaches, and propose a novel approach, using *atomic triples*, that overcomes them.

The specification and verification of concurrent program modules is a difficult problem. When concurrent threads work with shared data, the resulting behaviour can be complex. Two abstractions provide useful simplifications: that operations effectively act at distinct times; and that operations effectively act on disjoint resources. Programmers work with sophisticated combinations of time and data abstractions. In contrast, existing reasoning techniques tend to be limited to one or other abstraction.

Consider the following implementation of a concurrent counter:[3]

```
function read(x) {
  r := [x];
  return r;
}
```

```
function incr(x) {
  do {
    r := [x];
    b := CAS(x, r, r + 1);
  } while (b = 0);
}
```

```
function wkincr(x) {
  r := [x];
  [x] := r + 1;
}
```

The implementation provides an operation, `read`, that returns the current value of the counter, and two operations, `incr` and `wkincr`, that increment the value of the counter. The difference between `incr` and `wkincr` is that `wkincr` may not behave correctly if another thread is concurrently incrementing the counter (but is potentially faster otherwise[4]).

[3] We assume that memory read, write and compare-and-swap (CAS) operations are atomic.

[4] In a quick and dirty experiment, `wkincr` was around 60% faster.

Sequential Specification. How do we specify such a counter? A good start would be to give a sequential specification using Hoare triples, such as:

$$\{\mathsf{C}(\mathtt{x}, n)\}\ \mathtt{read(x)}\ \{\mathsf{C}(\mathtt{x}, n) \wedge \mathsf{ret} = n\}$$
$$\{\mathsf{C}(\mathtt{x}, n)\}\ \mathtt{incr(x)}\ \{\mathsf{C}(\mathtt{x}, n+1)\}$$
$$\{\mathsf{C}(\mathtt{x}, n)\}\ \mathtt{wkincr(x)}\ \{\mathsf{C}(\mathtt{x}, n+1)\}$$

In these specifications, the data representation used by the implementation is abstracted by the predicate $\mathsf{C}(\mathtt{x}, n)$, which denotes a counter at address $\mathtt{x}$ with value n.

Abstract Disjoint Specification. The above specification captures the sequential behaviour, but says nothing about concurrency. One way to specify concurrent behaviour is to use *disjointness*: each operation acts on specific resources, and threads that operate on disjoint resources do not interfere, so their effects can be combined. *Concurrent separation logics* [5] embody this principle in the form of a disjoint concurrency rule:

$$\frac{\{P_1\}\ \mathbb{C}_1\ \{Q_1\} \quad \{P_2\}\ \mathbb{C}_2\ \{Q_2\}}{\{P_1 * P_2\}\ \mathbb{C}_1 \parallel \mathbb{C}_2\ \{Q_1 * Q_2\}}$$

Assertions in separation logic describe resources, and confer ownership of those resources. The separating conjunction $P_1 * P_2$ describes the disjoint combination of the resources of P_1 and P_2. The *concurrent abstract predicates* (CAP) [2] approach supports specifications with abstract disjoint resources. The implementation of these abstract resources can involve *shared* resources. The abstract disjoint resources may be split, which effectively allows concurrent manipulation at the abstract level.

Treating the abstract predicate $\mathsf{C}(\mathtt{x}, n)$ as a resource, we could use the above sequential specification as a concurrent one. However, for multiple threads to use the counter, they would have to transfer the resource between each other using some form of synchronisation. Such a specification effectively enforces sequential access to the counter. This is because the client has no mechanism for dividing the resource: in particular,

$$\mathsf{C}(\mathtt{x}, n) \implies \mathsf{C}(\mathtt{x}, n) * \mathsf{C}(\mathtt{x}, n)$$

does not hold.

Bornat *et al.* [1] introduced *permission accounting* to separation logic, which allows resources to be divided. With fractional permissions, this is achieved by associating a fraction in the interval $(0, 1]$ with resources; resources may be subdivided by splitting this fraction. For instance, we may associate fractions with our counter resources to achieve:

$$\mathsf{C}(\mathtt{x}, n, \pi_1 + \pi_2) \iff \mathsf{C}(\mathtt{x}, n, \pi_1) * \mathsf{C}(\mathtt{x}, n, \pi_2)$$

We can modify our counter specification to give concurrent read access:

$$\begin{gathered}\{\mathsf{C}(\mathtt{x}, n, \pi)\}\ \mathtt{read}(\mathtt{x})\ \{\mathsf{C}(\mathtt{x}, n, \pi) \wedge \mathsf{ret} = n\} \\ \{\mathsf{C}(\mathtt{x}, n, 1)\}\ \mathtt{incr}(\mathtt{x})\ \{\mathsf{C}(\mathtt{x}, n+1, 1)\} \\ \{\mathsf{C}(\mathtt{x}, n, 1)\}\ \mathtt{wkincr}(\mathtt{x})\ \{\mathsf{C}(\mathtt{x}, n+1, 1)\}\end{gathered}$$

Note that we require full permission (1) in order to perform either increment operation. This means that only concurrent reads are permitted; updates must be synchronised with all other accesses. If only partial permission were necessary, then the specification for `read` would be incorrect, since it could no longer guarantee that the value being read matched the resource it had.

To specify concurrent increments, we can instead change how we split counter resources:

$$\mathsf{C}(\mathtt{x}, n_1 + n_2, \pi_1 + \pi_2) \iff \mathsf{C}(\mathtt{x}, n_1, \pi_1) * \mathsf{C}(\mathtt{x}, n_2, \pi_2) \qquad (n_1, n_2 \in \mathbb{N})$$

Now the resource $\mathsf{C}(\mathtt{x}, n, \pi)$ no longer asserts that the value of the counter is n (except if $\pi = 1$). Rather, it should be seen as accounting for a contribution of n to the value of the counter; other threads may also have contributions. We then specify our counter operations as:

$$\begin{gathered}\{\mathsf{C}(\mathtt{x}, n, \pi)\}\ \mathtt{read}(\mathtt{x})\ \{\mathsf{C}(\mathtt{x}, n, \pi) \wedge \mathsf{ret} \geq n\} \\ \{\mathsf{C}(\mathtt{x}, n, 1)\}\ \mathtt{read}(\mathtt{x})\ \{\mathsf{C}(\mathtt{x}, n, 1) \wedge \mathsf{ret} = n\} \\ \{\mathsf{C}(\mathtt{x}, n, \pi)\}\ \mathtt{incr}(\mathtt{x})\ \{\mathsf{C}(\mathtt{x}, n+1, \pi)\} \\ \{\mathsf{C}(\mathtt{x}, n, 1)\}\ \mathtt{wkincr}(\mathtt{x})\ \{\mathsf{C}(\mathtt{x}, n+1, 1)\}\end{gathered}$$

This specification at last allows concurrent reads and increments, although `wkincr` must still be synchronised with the other operations. The specification also fails to account for the fact that sequenced reads will never see decreasing values of the counter (since the contribution is not changed and provides the only lower bound). We could proceed to describe a more elaborate permission system that allows `wkincr` in the presence of reads, and to extend the predicate to record the last known value as a lower bound for reads. This would give us a more useful, if somewhat cumbersome, specification. Yet it would not handle a particularly important use case: synchronisation.

A ticketed lock exemplifies the use of counters for synchronisation. The lock is acquired by first incrementing one counter to acquire a notional ticket resource.[5] When the value of the second counter agrees with this ticket, the resource can be exchanged for the resource guarded by the lock. This resource is relinquished when the unlock operation is called, which increments the second counter.[6]

Such a ticketed lock has been verified using CAP [2]. However, the proof depends on the atomicity of the underlying operations in order to synchronise

[5] This requires that the increment operation should return the value of the counter, which we chose not to do for simplicity.

[6] The increment to the second counter will only be performed by the thread holding the lock, so `wkincr` is handy here.

access to shared resources. The proof, therefore, would not work with any of our abstract specifications, since they simply do not embody the necessary atomicity.

Atomic Specification. *Atomicity* is the abstraction that an operation takes effect at a single, discrete instant in time. The concurrent behaviour of atomic operations is equivalent to some sequential interleaving of the operations. *Linearisability* [4] is a correctness condition which specifies that the operations of a concurrent module appear to behave atomically. A client can then use these as if they were simple atomic operations, for example to implement a ticketed lock.

With linearisability, each operation is given a sequential specification, and the operations are asserted to behave atomically *with respect to each other*. Given our sequential specification for the counter, is our implementation linearisable? If we only consider `read` and `incr`, then the answer is 'yes'; however, adding `wkincr` breaks linearisability. The problem with `wkincr` is that, for instance, two concurrent calls could result in the counter only being incremented once. This is not consistent with atomic behaviour.

The essence of the problem is that we only really want `wkincr` to be called in a concurrent context where there are no other increments. In such a case, it would appear to behave atomically. However, by itself the sequential specification cannot express this constraint. To do so, we introduce a new form of specification, using *atomic triples*.

An atomic triple has the following form:

$$\mathbb{V}x \in X.\, \langle P(x) \rangle\ \mathbb{C}\ \langle Q(x) \rangle$$

Superficially, this can be read as "$\mathbb{C}$ atomically updates $P(x)$ to $Q(x)$ (for arbitrary $x \in X$)". What it actually means is a bit more subtle.

An implementation of the specification may assume that the assertion $P(x_0)$ holds initially (for some $x_0 \in X$). It must tolerate interference from the environment updating $P(x)$ to $P(x')$ (for any $x, x' \in X$). It is at liberty to update the state, providing that it preserves $P(x)$ (for the current value of x), until it updates it to $Q(x)$. After this update $Q(x)$ is no longer available to the implementation (another thread may be using it). Finally, the implementation cannot terminate without having update $P(x)$ to $Q(x)$ at some point.

Using the atomic triple, we can specify the counter as:

$$\begin{gathered}
\mathbb{V}n.\, \langle \mathsf{C}(\mathtt{x}, n) \rangle\ \mathtt{read}(\mathtt{x})\ \langle \mathsf{C}(\mathtt{x}, n) \wedge \mathsf{ret} = n \rangle \\
\mathbb{V}n.\, \langle \mathsf{C}(\mathtt{x}, n) \rangle\ \mathtt{incr}(\mathtt{x})\ \langle \mathsf{C}(\mathtt{x}, n+1) \rangle \\
\langle \mathsf{C}(\mathtt{x}, n) \rangle\ \mathtt{wkincr}(\mathtt{x})\ \langle \mathsf{C}(\mathtt{x}, n+1) \rangle
\end{gathered}$$

Intuitively, the first two specifications state the value of the counter will be read and incremented atomically, even in the presence of concurrent updates by the environment that change the value of the counter — since the value n is bound by $\mathbb{V}$. However, the environment must preserve the counter, e.g. it cannot deallocate it. The last specification means that `wkincr(x)` will atomically update the counter from n to $n+1$, as long as the environment guarantees that

the shared counter will not change value before the atomic update — since the value of n is not bound by $\mathbb{\forall}$.

This counter specification is strong: a client can derive the abstract disjoint specifications from it. Moreover, it is strong enough to support synchronisation: the correctness of a ticketed lock can be justified from this counter specification.

There are several benefits to using atomic triples for specifying concurrent modules. Atomic triples are expressive enough to enforce obligations on both the client and the implementation. By contrast, CAP specifications tend to unduly restrict the client (*e.g.* a counter specification cannot be used for synchronisation), while linearisability specifications tend to unduly restrict the implementation (*e.g.* a counter cannot provide a `wkincr` operation).

Atomic triples specify operations with respect to an abstraction (*e.g.* $\mathsf{C}(x, n)$), which means that each operation can be verified independently. This makes it possible to extend modules with new operations without having to verify the existing operations again. Linearisability, by contrast, is a whole module property: adding new operations (*e.g.* `wkincr`) can break the linearisability.

In [6], we introduce a generalised version of the atomic triple that can combine atomicity with resource transfer. For example, we can specify an operation that reads the value of the counter into a buffer; the read happens atomically, but the write to the buffer does not, and so ownership of the buffer is transferred between the client and implementation. This is not possible with traditional linearisability, although Gotsman and Yang [3] have proposed an extension of linearisability that supports ownership transfer.

Conclusions. We have introduced atomic triples as a way to specify abstract atomic commands. They allow us to overcome the limitations of linearisability by describing precisely the context that they can be used, effectively specifying when a client can use them. Moreover, they improve on CAP approaches by allowing the access of shared resources concurrently, instead of relying on primitive atomic commands.

In [6], we present TaDA, a program logic for Time and Data Abstraction, which extends CAP with rules for deriving and using atomic triples. We apply TaDA to a number of examples, proving implementations of a lock, multiple-compare-and-swap library, and double-ended queue against atomic specifications. Each example builds on the specification of the previous one, demonstrating the modularity of the approach.

References

1. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: POPL. pp. 259–270 (2005)
2. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent abstract predicates. In: ECOOP. pp. 504–528 (2010)
3. Gotsman, A., Yang, H.: Linearizability with ownership transfer. In: CONCUR. pp. 256–271 (2012)

4. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (Jul 1990)
5. O'Hearn, P.W.: Resources, concurrency, and local reasoning. Theor. Comput. Sci. 375(1-3), 271–307 (Apr 2007)
6. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: TaDA: A logic for time and data abstraction. In: ECOOP (2014), to appear