Subjective Concurrent Protocol Logic

Work in Progress

Aleksandar Nanevski
IMDEA Software Institute
aleks.nanevski@imdea.org

Ruy Ley-Wild
LogicBlox
rleywild@gmail.com

Ilya Sergey
IMDEA Software Institute
ilya.sergey@imdea.org

Germán Andrés Delbianco
IMDEA Software Institute
german.delbianco@imdea.org

1. Motivation and Outline

Modularity is a curse of logic-based verification of programs with shared-memory concurrency. Its challenge is manifold: one should be able to check the safety properties of a thread running in isolation, as well as to specify, what are the "safe" states of a particular data structure and check them separately, abstracting from the implementation of other components of a system. The last decade witnessed a blast of approaches in attempt to tackle the modularity problem by employing different combinations of *concurrent separation logic* for spatial reasoning about safety properties and *rely/guarantee* for specifying temporal invariants of programs running in parallel. Expressive enough, all these techniques pay their price by either dealing with too low-level model and forcing a client of a logic to give verbose specifications, or by boiling the logical reasoning down to the underlying semantic model.

In our work, we take a different view at the model of concurrent computations and provide a simple yet powerful logical framework, enabling reasoning about concurrently running independent modules *at the level of specifications*. The key novelty is the way to specify the module protocols in the form of communicating state transition systems (STS) that describe valid program states and the transitions the program is allowed to make. In this talk we will show how the reasoning in terms of STS makes it simple to crystallise the behavioural invariants of a module, and provide an *entanglement* operator to build large systems from an arbitrary number of modules.

2. Key Ideas of the Approach

In this work, we follow the idea to model a concurrent program representing it as a system of inter-communicating state transition systems (STS). For example, a mutual exclusion protocol followed by programs that use just one lock, may abstractly be represented by an STS. More precisely, the lock itself may be in one of two states (*locked* or *unlocked*), but any individual thread t may need to make a finer distinction of the locked states: it may need to differentiate between the lock being owned by t itself, or by some other thread running concurrently with t.

By abstraction at the level of specification in this particular example, we consider the ability of a formal system to represent a mutual exclusion protocol in the form of an STS, while ignoring all the information about the low-level particulars of the locking mechanism (e.g., is it a spin lock, or a ticketed lock). As the number of locks that one program uses grows, the complexity of the STS that specifies the program behaviour grows exponentially. By modularity at the level of specification, we consider the ability of a formal system to ignore the those states and transitions of the STS, that may be irrelevant for the verification of a particular program.

Below, we list the key ingredients of our verification approach named *Subjective Concurrent Protocol Logic* (SCPL), wich we are going to present in the talk.

Concurroids, States and Transitions We introduce a notion of *concurroid* as a main unit of decomposition in model of a concurrently-running program. Concurroids are *communicating* STS's as we endow them with the ability to exchange *ownership* of portions of their states. For example, a concurroid for a lock protecting a shared resource in the Owicki-Gries Resource Invariants (RI) or Concurrent Separation Logic (CSL) style, upon making a transition from an unlocked to a locked state, *gives away* the heap belonging to a shared resource by means of an *external transition* (i.e., a transition towards an outside world). Dually, upon making a transition from a locked to an unlocked state, it *accepts* a heap from the outside world.

Transitions are relations on states of a concurroid describing how the state of concurroid may change. Each concurroid has three different transitions: (1) *internal*, (2) *acquire*, and (3) *release* transition. The internal transition describes the modifications to the contents of the concurroid state, such as, e.g., changing the value stored into some pointer of the private state. Acquire and release transitions are indexed by a heap, and describe how the concurroid can receive (dually send) a chunk of heap from/to the outside world.

Entanglement In order to build a complex model from multiple concurroids by composing them, we present the *entanglement* operator. The operator combines two smaller concurroid STS's by connecting their *dually-polarised* external transitions. The entanglement operator is tightly connected to specification-level modularity, as it can be seen as an STS equivalent of separating conjunction $*$ from CSL. In particular, we will illustrate how the usual ownership transfer from CSL, may be represented as a communication between a concurroid for private state, with the concurroid for locking (and/or allocation).

Atomic actions Concurroids are just abstract models, specifying how particular part of a concurrently-running system should behave. We bridge concurroids with the actual axiomatic semantics of a simple imperative programming language by supplying each concurroid with a number of *atomic actions* (sometimes referred to just as *actions*). Actions are the primitive objects for programming in the language of SCPL. They perform *atomic* steps from state to state, and have the lowest granularity. A major yet justified design restriction is that *all* actions should look like primitive *read*, *write* of *CAS*. Although our framework does not enforce this restriction automatically, in our development and examples we follow it faithfully. The reason for this restriction is natural: the listed primitives are well-understood and their power for establishing a consensus is known. Moreover, this level of atomicity makes them easy to implement in hardware.

An action may also be idle. Additionally, actions may return a value of some given type A, as well as manipule the auxiliary state, and perform ownership transfer (i.e., realign the boundaries between *self*, *joint* and *other* components). An action is always associated with some concurroid V in the following sense. If an action modifies state s into state s', then the transition (s, s') is internal for the concurroid V. In our formalisation, we also have actions that are associated with $acquire$ and $release$ transitions. However, these are not used for programming, but only for building other $internal$-related actions of larger, *composite* concurroids.

Subjectivity and Partial Commutative Monoids Our framework follows the subjective approach of the previous work on Subjective Concurrent Separation Logic (SCSL) (Ley-Wild and Nanevski 2013), in that SCPL explicitly divides the states of each transition system into three components *self* (belonging exclusively to the thread that is being specified), *joint*, and *other* (belonging exclusively to the environment threads that run concurrently with the thread being specified). Moreover, the self and other components are not arbitrary, but have to satisfy the algebraic requirements of a Partial Commutative Monoid (PCM).

As argued in the previous work on SCSL, the subjective view (i.e., the presence of the other, which is not present in any related work) allows for a compositional specification in the presence of auxiliary state. It also simplifies our logic, as it enables a *uniform* treatment of the interaction between a thread and an environment that follow a certain protocol. For example, SCPL *does not* require separate rely and guarantee transitions of its state transition systems. Essentially, each transition of the system can be viewed as being either, simply by *transposing* the partition of the state on which it operators (i.e., exchanging the *self* and *other* parts).

Implementation by shallow embedding into Coq We proved the soundness of our logic with respect to the Brookes-style denotational semantics. Moreover, we implemented the semantics, metatheory, logic itself and a number of examples as a shallow embedding in Coq, which made it possible for us to program and prove directly with the semantic objects, thus lifting SCPL to a full-blown programming language and verification system with higher-order functions, abstract types, abstract predicates, and a module system. We also gained a powerful dependently-typed λ-calculus, which we used to formalize all semantic definitions and metatheory.

3. Presentation Plan

In our talk, we are planning to start from building a basic intuition about concurroids, transitions and actions using the Hello World example of the concurrent program verification—a CAS-based spin-lock. Alas, this example is not very illustrative, as it does not fully show all the components of the framework, namely, internal transitions and advanced use of PCMs to account for subjective view and resource splitting. In order to demonstrate the use of the framework on a more advanced concurrent protocol, we design a concurroid for a ticketed lock. As a reference implementation, we use the one from the work on Concurrent Abstract Predicates (Dinsdale-Young et al. 2010).

```
lock = {                          unlock = {
  n := INCRNXT;                     INCROWN;
  while (!TRY(n)) skip;           }
}
```

The three (supposedly atomic) commands `INCRNXT`, `TRY(n)` and `INCROWN` have the following operational meaning:

```
INCRNXT  = { tmp := next; next++; return tmp; }
TRY(n)   = { return (n = owner); }
INCROWN  = { owner++; }
```

The pointers `owner` and `next` point to natural numbers, describing the state of the ticketed lock system. The intuition is similar to the organisation of the service in a bakery: `next` corresponds to the current ticket available at the dispenser (but not yet taken!), whereas `owner` indicates a ticket number, an owner of which is currently being served (or just about to be).

What are the primary resources that a ticketed lock deals with? Those are tickets, of course! Some of them can be taken by a current thread, some by others. If we make an assumption that each ticket is thrown away right after its owner has been served, we will see a simple invariant: all tickets in the system are from the range $[n_1, n_2)$, where n_1 and n_2 are natural numbers, corresponding to the values of `owner` (currently being served or about to be) and `next` (next ticked to be issued), hence non-inclusion on the right side of the range.

With this intuition in mind, we will proceed by constructing the concurroid and defining its transitions. We will culminate by defining three atomic read/write/CAS-like looking actions for the ticketed lock concurroid. These action will indeed correspond exactly to the above mentioned commands: `INCRNXT`, `TRY(n)` and `INCROWN`. An important thing to note is the way the atomic actions are going to be used. Of course, we are going to employ them to implement the reference algorithm, but other applications are possible and in any case, the actions will preserve the defined coherent structure of the concurroid and respect the transitions.

Out of curiosity, we will discuss other possible non-orthodox usages of the same concurroid. For instance, one can think of an "abusive" lock algorithm, in which a thread fetches several tickets consequently (by running `INCRNXT`) and then distributes them among its children.[1] In fact, this scenario is perfectly valid for the concurroid, and the actions, designed naturally from the structure of the concurroid, make it possible, too. This last observation contributes to the idea of thinking about *concurrent protocols* just as of *programs* in terms of communicating independent STS's. Implementing *well-known* concurrency patterns (and, therefore, verifying them) would be one possible use of this mode, but, as long as the defined invariants are preserved, other interesting (and safe!) protocols may be implemented in the same terms.

Depending on the time constraints, we are planning to sketch a guide for designing a concurroid for a Bornat-style Readers-Writers protocol (Bornat et al. 2005).

References

Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. Permission accounting in separation logic. In *POPL*, 2005.

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent Abstract Predicates. In *ECOOP*, 2010.

Ruy Ley-Wild and Aleksandar Nanevski. Subjective auxiliary state for coarse-grained concurrency. In *POPL*, 2013.

[1] One of the authors witnessed this situation in a real bakery.