

Specifying and verifying liveness properties of QLOCK in CafeOBJ

Norbert Preining, Kazuhiro Ogata, and Kokichi Futatsugi

Japan Advanced Institute of Science and Technology
Research Center for Software Verification
Nomi, Ishikawa, Japan
{preining,ogata,futatsugi}@jaist.ac.jp

Abstract

We provide an innovative development of algebraic specifications and proof scores in CafeOBJ of QLOCK’s safety and liveness properties. The particular interest of the development is two-fold: Firstly, it extends base specifications in order-sorted and rewriting logics to a meta-level, which requires behavioral logic, thus using the three logics together to achieve the proofs. Secondly, we use a search predicate and covering state patterns that allow us to prove the validity of a property over all possible one-step transitions, by which safety and liveness properties in the base and meta-level can be proven.

1998 ACM Subject Classification D.2.4 Formal methods, Correctness proofs

Keywords and phrases fairness, liveness, CafeOBJ, verification, algebraic specification

1 Introduction

QLOCK, an abstract version of Dijkstra’s binary semaphore, is a protocol to guarantee exclusive access to a resource. Besides the initial specification and verification in CafeOBJ (see for example [2]), it saw implementations in Coq and Maude. Most of these specifications only consider safety properties, in the current case the mutual exclusion property, that no two agents will have access to the resource at the same time. However, liveness properties are normally left open. These properties ensure that ‘there is progress’. In our particular case, they ensure that agents do not block out other agents from acquiring access to the resource.

1.1 The QLOCK protocol

The QLOCK protocol regulates access of an arbitrary number of agents to a resource by providing a queue (first-in-first-out list). Agents start in the *remainder section*, henceforth indicated by *rs*. If an agent wants to use the resource, it puts a unique identifier into the queue, and by this transitions into the *waiting section* (*ws*). In the waiting section, an agent checks the top of the queue. If it is its unique Id, the agent transitions into the *critical section* (*cs*), during which the agent can use the resource. After having finished with the resource usage, the agent removes the head of the queue and transitions back into *rs*.

1.2 Verification properties

The basic safety property of QLOCK is the *mutual exclusion property* (*mp*):

► **Property 1 (mutual exclusion property).** At any time, at most one agent is in the critical section.

While this is the most important property for safety concerns, it does not guarantee that an agent wanting to use the resource ever gets the chance to use it. To guarantee this, we

define two liveness properties: The first concerns the transition from **ws** to **cs**, and is called the *progress property* (**pp**). This property has already been discussed in [7] as *lockout freedom property*.

► **Property 2 (progress property)**. An agent that has entered into **ws** (waiting section), i.e., has put his **Id** into the queue, will eventually progress to the top of the queue and gain access to the resource, i.e., transition into **cs** (critical section).

The last one concerns the transition from **rs** to **ws**, called *entrance property* (**ep**):

► **Property 3 (entrance property)**. An agent will eventually transition from **rs** to **ws**.

It might sound counter-intuitive that the entrance property should hold for each agent at all times, but what we are effectively proving is that any given arbitrary finite sequence of transitions (where the agent might not want to enter at all) can be extended to an infinite fair transition sequence, and within this transition sequence the property holds. Reflecting back to the finite transition sequence, we obtain that either the agent can enter the queue or the execution stops. Thus, this last property is conceptually different from the first two, as it requires an additional assumption on the fairness of the transition sequence, see Section 3.2.

1.3 Observational transition systems

Observational Transition System, henceforth **OTS**, is a modeling scheme for arbitrary systems that tries to describe the system by a set of observers. Like a state machine, an **OTS** features transitions, describing the change of observation values due to activity of/in the system.

OTSs have been successfully employed to specify and verify various protocols in **CafeOBJ** [5]. We are convinced that the usage of observations, that is, not inherent properties of the implementation, but abstract properties of the system exhibited by the change of observations, allow for algebraic specifications with good properties, in particular verification by induction on the reachable state space.

In our specific case, it means that we will describe the system by the observations of the states, i.e., in which section agents are, as well as the state of the queue.

1.4 Verification by induction and exhaustive search

While we cannot give a full description of the theory behind verification by induction of an **OTS** system, the following should explain the basic idea.

Verification of properties of an **OTS** is often done by induction on reachable states. That is, we show that a certain property (**invprop**) holds in the set of initial states, characterized by **init**. Furthermore, as we proceed through transitions (state changes), the **invprop** is preserved.

But to show liveness properties, considering only invariant properties on states is not enough. We thus use an extended method that does inductive proofs on the reachable state space, and in parallel proves properties (**transprop**) on all transitions between reachable states. To be a bit more specific, assume that $S \Rightarrow SS$ is a transition from one state pattern term S to SS . We show that if **invprop**(S) holds, then also **invprop**(SS) (the induction on reachable states), but also that for this transition **transprop**(S, SS) holds.

Both of these are done with **CafeOBJ**'s builtin search predicate, which exhaustively searches and tests all possible transitions from a given state pattern. The concepts introduced here are an extension and generalization of *transition invariants* [8], details in a forthcoming publication.

2 Specification

We are using CafeOBJ as specification and verification language. CafeOBJ is a many- and order-sorted algebraic specification language from the OBJ family, related to languages like CASL and Maude. CafeOBJ allows us to have both the specification and the verification in the same language. It is based on powerful logical foundations (order-sorted algebra, hidden algebra, and rewriting logic) with an executable semantics [3, 5, 6].

2.1 Specification and verification of invariant properties

As mentioned above, we are building upon a previously obtained specification and verification of QLOCK [2]. We will give only the list of defined modules, and refer the reader to the web page¹ for the full code. We cannot give a full introduction to the CafeOBJ language here, but users acquainted to Maude will find it easy to read the code.

Skipping the definitions of labels, agent identifiers and the queue, the following code specifies agent observers, codifying the agent id and the current section of the agent in one term. The meaning of the term $lb[A]:S$ is that the agent A is in section S :

```
mod! AOB {protecting(LABEL) protecting(AID) [Aob]
  op (lb[_]:_) : Aid Label -> Aob {constr} . }
```

The state of the whole OTS is described as a pair of a queue and a set of agent observers, where the pairing is achieved by the $\$$ construct (CafeOBJ allows nearly arbitrary syntax):

```
mod! STATE{ protecting(AID-QUEUE)
  protecting(SET(AOB{sort Elt -> Aob})*{sort Set -> Aobs})
  [State] op _$_ : Qu Aobs -> State {constr} . }
```

The transitions are defined by transition rules over state patterns:

```
mod! WaitTrans { protecting(STATE)
  trans[wt]: (Q:Qu $ ((lb[A:Aid]: rs) AS:Aobs))
    => ((Q & A) $ ((lb[A] : ws) AS)) . }
mod! TryTrans { protecting(STATE)
  trans[ty]: ((A:Aid & Q:Qu) $ ((lb[A]: ws) AS:Aobs))
    => ((A & Q) $ ((lb[A]: cs) AS)) . }
mod! ExitTrans {protecting(STATE)
  trans[ex]: ((A1:Aid & Q:Qu) $ ((lb[A2:Aid]: cs) AS:Aobs))
    => (Q $ ((lb[A2] : rs) AS)) . }
```

Based on the above specification, it is possible to provide a *proof score*, i.e., a program in CafeOBJ that verifies the mutual exclusion property *mp*.

2.2 Original progress property

The property that an agent being in the queue, i.e., in *ws*, will eventually progress to the *cs* state and gain access to the resource, was originally not part of the verification, but it turned out that it can be shown under the sole assumption that the number of agents is finite (but arbitrary). The core idea is to keep track of the position of an agent in the queue.

During the lift to the meta-level, we will re-prove this fact, and thus will not go into details of the original verification.

¹ <http://www.preining.info/blog/cafeobj/>

3 Going to the meta-level

To verify the last property, *ep*, operational considerations alone do not suffice. On the level of observers, we cannot guarantee that an agent will ever enter the queue, since we have no control over which transitions are executed by the system. To discuss (verify) this property, we have to assume a certain meta-level property, in this case the fairness of the transition sequence. A similar approach has been taken in [4] for the Alternating Bit Protocol, where fair *event mark streams* are considered.

3.1 Relation to other concepts of fairness

The methodology of using OTS in CafeOBJ has been strongly influenced by UNITY [1], which provides an *ensures* operator allowing to model fairness.

Another approach to the concept of fairness is taken by LTL logic [9], where two types of fairness, strong and weak, are considered, referring to *enabled* and *applied* state of transitions.

3.2 Transition sequence

Modeling fairness requires recurring to a meta-assumption, namely that the sequence of transitions is fair, i.e., every instance of a transition appears infinitely often in the sequence. In our case we wanted to have a formalization of this meta-assumption that can be expressed with the rewriting logic of CafeOBJ.

The approach we took models transition sequences using behavioral specification with hidden algebra [4], often used to express infinite entities. Note that we are modeling the transition sequence by an infinite stream of agent ids, since the agent uniquely defines the instance of transition to be used, depending on the current state of the agent:

```
mod* TRANSSEQ { protecting(AID)
  * [ TransSeq ] *
  op ( _ & _ ) : Aid TransSeq -> TransSeq . }
```

The transition sequence is then used to model a *meta-state*, i.e., the combination of the original state of the system as specified in the base case, together with the list of upcoming transitions:

```
mod! METASTATE { protecting(STATE + ... ) [MetaState]
  op _ ^ _ : State TransSeq -> MetaState {constr} . ... }
```

In the same way, transitions from the base case are lifted to the meta-level. We give the meta-version for the transition into *ws* as an example exhibiting how the state changes while transitions are used up from the transition sequence.

```
mod! MWT {protecting(METASTATE)
  trans [meta-wt]:
    ( (Q:Qu $ ((lb[A:Aid]: rs) AS:Aobs)) ^ (A:Aid & T:TransSeq))
=> ( (Q & A) $ ((lb[A]: ws) AS) ^ T ) . }
```

To express the fairness condition, we recurred to an equivalent definition, namely that every finite sequence of agent ids can be found as a subsequence of the transition sequence, rephrased here in an indirect way:

```
eq ( find ( Q:Qu, T:TransSeq ) = empQ ) = false .
```

3.3 Verification of properties

By defining (and computing) the wait-time of an agent as the number of meta-transitions (or alternatively, the number of steps in a computation according to the transition sequence), we were able to describe the two liveness properties mentioned above. This was done by proving two invariant properties, namely: (1) If an agent does not change its section during a transition, then its wait-time is decreasing. (2) If the wait-time of an agent reaches 0, then it will change its section. These two invariant properties together show that both the progress property **pp** and the entrance property **ep** hold. Note that these properties do not only depend on the states, but properties that hold on the combination of states and transitions. With the same method, we re-proved the *mutual exclusion property*, as well as an extended version of the *progress property*: The original verification requires that there is a finite number of agents. In contrast, if we assume fairness of the execution sequence, even in the case of an infinite number of agents, both *progress* and *entrance* properties hold.

4 Discussion and conclusion

We have to note that what we called here *progress property* has already been shown in different settings [7]. The key contribution is the extension to the *entrance property*, meaning that an agent always gets a chance to enter the queue. In addition, we could extend the proof of the progress property to infinitely many agents. The current work also serves as an example of reflecting meta-properties into specifications, allowing for the verification of additional properties.

Assuming a meta-level fairness property to prove liveness properties of the specification might be considered a circular argument, but without regress to meta-level fairness, *no* proof of the entrance property can be achieved. Keeping this in mind, our goal is to provide a reasonable simple and intuitive definition of fairness on the meta-level, that can be used for verification of the necessary properties, similar to any axiomatic approach where trust is based on simple axioms. We are confident that this approach proves successful in other circumstances like the mentioned Alternating Bit Protocol.

References

- 1 K.M. Chandy and J. Misra. *Parallel program design - a foundation*. Addison-Wesley, 1989.
- 2 K. Futatsugi. Generate & check methods for invariant verification in CafeOBJ. JAIST Research Report IS-RR-2013-006, 2013. <http://hdl.handle.net/10119/11536>.
- 3 K. Futatsugi, D. Găină, and K. Ogata. Principles of proof scores in CafeOBJ. *Theor. Comput. Sci.*, 464:90–112, 2012.
- 4 J.A. Goguen and K. Lin. Behavioral verification of distributed concurrent systems with BOBJ. In *QSIC*, pages 216–. IEEE Computer Society, 2003.
- 5 Sh. Iida, J. Meseguer, and K. Ogata, editors. *Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi*, volume 8373 of *LNCS*. Springer, 2014.
- 6 J. Meseguer. Twenty years of rewriting logic. *J. Log. Algebr. Program.*, 81(7-8):721–781, 2012.
- 7 K. Ogata and K. Futatsugi. Proof score approach to verification of liveness properties. *IEICE Transactions*, 91-D(12):2804–2817, 2008.
- 8 A. Podelski and A. Rybalchenko. Transition invariants. In *LICS*, pages 32–41. IEEE Computer Society, 2004.
- 9 V. Rybakov. Linear temporal logic with until and next, logical consecutions. *Annals of Pure and Applied Logic*, 155(1):32–45, 2008.