Title	Generate & Check Methods for Invariant Verification in CafeOBJ
Author(s)	Futatsugi, Kokichi
Citation	Research report (School of Information Science, Japan Advanced Institute of Science and Technology), IS-RR-2013-006: 1-31
Issue Date	2013-11-27
Type	Technical Report
Text version	publisher
URL	http://hdl.handle.net/10119/11536
Rights	
Description	リサーチレポート（北陸先端科学技術大学院大学情報科学研究科）

Generate & Check Methods for Invariant Verification in CafeOBJ

Kokichi Futatsugi
Research Center for Software Verification

2013年11月27日
IS-RR-2013-006

Generate&Check Methods for Invariant Verification in CafeOBJ

Kokichi Futatsugi

Research Center for Software Verification (RCSV)
Japan Advanced Institute of Science and Technology (JAIST)
1-1 Asahidai, Nomi, Ishikawa 923-1292, Japan
futatsugi@jaist.ac.jp

Abstract. Effective coordination of inference (*à* la theorem proving) and search (*à* la model checking) is one of the most important and interesting research topics in formal methods. We have developed several techniques for coordinating inference and search for verification with proof scores in CafeOBJ. The generate&check methods proposed in this paper are recent developments for invariant verification of this kind. The methods are based on (1) state representations as sets of observers, and (2) systematic generation of finite state patterns which subsume all possible infinite states.

This paper describes the generate&check methods and their theoretical foundation. The methods and theory are explained with a small but instructive example of mutual exclusion protocol. The explanation is intended to be self-contained, and includes necessary basics of the CafeOBJ language/system also.

1 Introduction

Constructing specifications and verifying them in upstream of software development are still the most important challenges in formal software engineering. It is because quite a few critical bugs are caused at the level of domains, requirements, and/or designs specifications. Proof scores are intended to meet this challenge [5, 6].

In proof score approach, an executable algebraic specification language (i.e. CafeOBJ [2] in our case) is used to specify systems and system properties, and a processor (i.e. rewrite engine or reducer) of the language is used as a proof engine to prove that the systems satisfy properties of interest. Proof plans are coded into proof scores, and are also written in the algebraic specification language. The proof scores are executed by the rewrite engine, and if everything is as expected, an intended proof has been successfully done. Logical soundness of this procedure is guaranteed by the fact that rewritings/reductions done by the rewrite engine is consistant with equational axioms of original specifications [7].

The concept of proof supported by proof scores is similar to that of LP [10]. Proof scripts written in tactic languages provided by proof assistants such as Coq [1] and Isabel/HOL [12] have similar nature as proof scores. However, proof scores are written uniformly with specifications in an executable algebraic specification language and can enjoy a transparent, simple, and powerful logical foundation based on equational and rewriting logic [7, 11].

Effective coordination of inference [1, 12] and search [3, 9] is important for making proof scores more effective and powerful, and we have developed several techniques for that [15, 6]. The generate&check methods proposed in this paper are recent developments for invariant verification. The methods are based on (1) state representations as sets of observers, and (2) systematic generation of finite state patterns which subsume[1] all possible infinite states. These two have been achieved based on the uniform and transparent logical foundation of proof scores [7].

The rest of the paper is organized as follows. Section 2 presents a mutual exclusion protocol QLOCK that is used to explain methods and theory throughout this paper. Section 3 presents a system specification of QLOCK with OTS in the CafeOBJ language. Section 4 explains transition systems and their invariant verification and presents a property specification of QLOCK. Section 5 presents a proof score for QLOCK using the generate&check methods. Explanations on the correctness of the methods are given throughout the section, and formal proofs are given in Section 5.3. Section 6 summarizes achievements, explains related works, and mentions future issues.

2 QLOCK: A Mutual Exclusion Protocol

Mutual excusion protocols can be described as follows:

> *Assume that many agents (or processes) are competing for a common equipment (e.g. a printer or a file system), but at any moment of time only one agent can use the equipment. That is, the agents are mutually excluded in using the equipment. A protocol (mechanism or algorithm) which can achieve the mutual exclusion is called "mutual exclusion protocol".*

A mutual exclusion protocol, called QLOCK, is realized by using a global queue (first in first out storage) of agent names (or identifiers) as follows.

- Each of unbounded number of agents who participates in the protocol behaves as follows:
  - If the agent wants to use the common equipment and its name is not in the queue yet, put its name into the bottom of queue.
  - If the agent wants to use the common equipment and its name is already in the queue, check if its name is on the top of the queue. If its name is on the top of the queue, start to use the common equipment. If its name is not on the top of the queue, wait until its name is on the top of the queue.

[1] Terms $t_1, \cdots, t_m$ are defined to **subsume** terms $t'_1, \cdots, t'_n$ iff for any t'_i $(i \in \{1, \cdots, n\})$ there exits t_j $(j \in \{1, \cdots, m\})$ such that t'_i is an instance[51] of t_j.

[51] The "instance" is defined formally in Definition 1.

  - If the agent finishes to use the common equipment, remove its name from the top of the queue.
- The protocol should start from the state where the queue is empty.

3 System Specification with OTS

OTS (Obervational Transition System) is a modeling scheme for transition systems (or state machines). A state of a transition system is identified as a collection of typed values given by *observers* (or observation operations). A state transition of the system is modeled as an *action* that defines the current state and the next state relation.

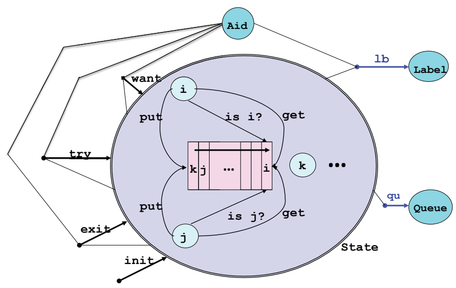

Fig. 1. Global view of QLOCK as an Observational Transition System

QLOCK is modeled in OTS as illustrated in Fig. 1.

For the generate&check methods, generations of finite state patterns that can subsume all the infinite states is a key procedure, and a state is assumed to be represented by an appropriate data structure (or configuration). Notice that this is different from the original OTS modeling scheme where there is no assumption on the structure of a state [13, 14].

3.1 `LABEL` and `AID`

For defining the state configuration of QLOCK, we first need the following two CafeOBJ modules `LABEL` and `AID`.[2]

```
-- three labels for indicating the status of each agent
mod! LABEL {
```

[2] ANNEX contains the complete CafeOBJ specification and proof score for QLOCK explained in this paper.

```
-- label literals and labels
[LabelLt < Label]
-- rs: remainder section, the agent's id is not in the queue yet
-- ws: waiting section, the agent's id is in the queue
-- cs: critical section, the agent is using the common equipment
ops rs ws cs : -> LabelLt {constr} .
eq (L1:LabelLt = L2:LabelLt) = (L1 == L2) .
}
-- agent identifiers
mod* AID {[Aid]}
```

A line starts with `--` is a comment line[3]. A keyword `mod` starts a **module** with the following module name (`LABEL` or `AID` in this case) and module body. A module body starts with { and ends with }. A character `!` or `*` following the keyword `mod` indicates that the module denote the unique initial (or standard) model or all the models of the module respectively. **Sort** (or type) names are declared between `[` and `]`. Symbol `<` indicates that sorts in the left hand side are **subsorts** (i.e. subsets) of the sort in the right hand side. The keyword `ops` is the plural form of `op`, and starts a declaration of **operators** (or function names) of the same arity (i.e. sequence of argument sorts) and co-arity (i.e. value sort). Arity and co-arity comes before and after `->`. The juxtaposition of arity and co-arity is called **rank**. `ops` and `op` end with "`.`" Here `rs`, `ws`, and `cs` are declared to be operators with null arity and co-arity `LabelLt`. An operator with null arity is called **constant**. Several **operator attributes** can be declared by putting corresponding keywords between { and } after the co-arity. Attribute `constr` means **constructor** and indicates that the constants (or operators with null arity) `rs`, `ws`, and `cs` are constructors.

`eq` starts an **equation** declaration, and the left hand side and the right hand side are declared befor and after "`=`". An equation declaration should end with "`.`". The equation in the module `LABEL` is declaring that `(L1:LabelLt = L2:LabelLt)` is equal to `(L1 == L2)`. `L1:LabelLt` is an on-line **variable** declaration, and the declared variable `L1` of the sort `LabelLt` is effective until the end of the equation. *Literals* generally mean names that *literally* identify the objects the names denote. That is, different literals denote different objects. `_=_`[4] and `_==_` are built-in binary predicate defined on any sort S, and have a rank "S S `Bool`", that is "`LabelLt LabelLt Bool`" in this case. Both of `_=_` and `_==_` return `true` if two arguments are reduced to the same term by using all declared equations as left to right reduction (or rewriting) rules. But if the two reduced terms t_1 and t_2 are different, `_==_` returns `false`, but `_=_` returns (t_1 `=` t_2). This implies that for constants of sort `LabelLt` that do not have any other reduction rules, `_=_` checks the literal equality of the name. That is, `((rs = rs) = true)`, `((rs = ws)`

[3] A line starts with `**`, `-->`, or `**>` is also a comment line.

[4] Notice that the object level Boolean predicate and meta level CafeOBJ equality operator that composes an equation are represented by the same symbol `_=_` but are different; they are easly distinguished from context.

`= false`), etc. The module `AID` just declares that any set can be the set of sort `Aid`.

3.2 QUEUE

Parametrized generic queues are defined by the following module `QUEUE`.

```
-- queue (first in first out storage)
mod! QUEUE (X :: TRIV) {
-- elements and their queues, Elt comes from (X :: TRIV)
[Elt.X < Qu]
-- error elements and error queues
[Elt.X < Elt&Err]  [Qu < Qu&Err]
-- empty queue
op empQ : -> Qu {constr} .
-- assoicative queue constructors with id: empQ
op (_&_) : Qu Qu -> Qu {constr assoc id: empQ} .
op (_&_) : Qu&Err Qu&Err -> Qu&Err {constr assoc id: empQ} .
-- equality _=_ over Qu&Err
eq (empQ = (E:Elt & Q:Qu&Err)) = false .
ceq ((E1:Elt & Q1:Qu&Err) = (E2:Elt & Q2:Qu&Err))
    = ((E1 = E2) and (Q1 = Q2))
    if not((Q1 = empQ) and (Q2 = empQ)) .
-- head
op hd_ : Qu&Err -> Elt&Err .
eq hd(E:Elt & Q:Qu&Err) = E .
-- hd(empQ) is not defined intentionally, an error handling method
-- tail
op tl_ : Qu&Err -> Qu&Err .
eq tl(E:Elt & Q:Qu&Err) = Q .
-- tl(empQ) is not defined intentionally, an error handling method
}
```

A parameter declaration `(X :: TRIV)` is placed after the module name `QUEUE`. A built-in module `TRIV` is just a renaming of `AID` and defined as "`mod* TRIV {[Elt]}`". This implies that the parameter for `QUEUE` can be any set of objects. `ceq` starts a declaration of a conditional equation, and declares a condition (i.e. a Boolean term) after the keyword `if`.

3.3 OBS, SET, and STATE

A state of QLOCK is defined as a set of observers by the following three modules.

```
-- observers
mod! OBS {
pr(LABEL)
```

```
pr(QUEUE(AID{sort Elt -> Aid}))
-- there are two kinds of obserbers
[Obs]
-- queue observer
op (qu:_ ) : Qu -> Obs {constr} .
-- agent observer
op (lb[_]:_) : Aid Label -> Obs {constr} .
}
-- generic set
mod! SET(X :: TRIV) {
[Elt.X < Set]
-- empty set
op empty : -> Set {constr} .
-- assicative and commutative set constructor with identity empty
op (_ _) : Set Set -> Set {constr assoc comm id: empty} .
-- (_ _) is idempotent
eq E:Elt E = E .
}
-- a state is defined as a set of observers
mod! STATE {pr(SET(OBS{sort Elt -> Obs})*{sort Set -> State})}
```

`pr(_)` indicates a **protecting** importation, and declares to import a module without changing its model (or models). `QUEUE(AID{sort Elt -> Aid})` defines the module obtained by instantiating the parameter `X` of `QUEUE` by `AID` with the renaming of `Elt` to `Aid`. `QUEUE(AID{sort Elt -> Aid})` and `SET(OBS{sort Elt -> Obs})` denote sets of agent identifiers and sets of observers respectively. `*{sort Set -> State}` after `SET(OBS{sort Elt -> Obs})` defines the renaming of `Set` to `State`.

3.4 `WT`, `TY`, `EX`, and `QLOCKsys`

The QLOCK protocol is defined by the following three modules. The transition rule of the module `TY` indicates that if the top element of the queue is `A:Aid` (i.e. `(qu: (A:Aid & Q:Qu))`) and the agent `A` is at `ws` (i.e. `(lb[A:Aid]: ws)`) then `A` gets into `cs` (i.e. `(lb[A]: cs)`) without changing contents of the queue (i.e. `(qu: (A & Q))`). The other two transition rules can be read similarly. Notice that the module `WT`, `TY`, `EX` formulate the three actions explained in the Section 2 precisely and succinctly. `QLOCKsys` is just combining the three modules.

```
-- wt: want transition
mod! WT {pr(STATE)
trans[wt]:
     ((qu: Q:Qu)(lb[A:Aid]: rs) S:State)
     => ((qu: (Q & A))(lb[A]: ws) S) .
}
-- ty: try transition
```

```
mod! TY {pr(STATE)
trans[ty]:
      ((qu: (A:Aid & Q:Qu))(lb[A]: ws) S:State)
      => ((qu: (A & Q))(lb[A]: cs) S) .
}
-- ex: exit transition
mod! EX {pr(STATE)
trans[ex]:
   ((qu: (A1:Aid & Q:Qu))(lb[A2:Aid]: cs) S:State)
   => ((qu: Q)(lb[A2]: rs) S) .
}
-- system specification of QLOCK
mod! QLOCKsys{pr(WT + TY + EX)}
```

A declaration of a transition rule starts with `trans`, contains rule's name `[_]:`, current term and next term before and after `=>` respectively, and ends with ".". Notice that because a state configuration is a set (i.e. a term composed of associative, commutative, and idempotent binary constructors `(_ _)`) the second component of the left hand side `(lb[A:Aid]: rs)` of the rule `wt` can match any agent in a state. This implies that the transition rule `wt` can define unbounded number of transitions depending on the number of agents a state includes. The same holds for the rules `ty` and `ex`.

4 Property Specification and Invariants

A majority of systems and problems in many fields can be modeled with transition systems and their invariants. An **invariant** of a transition system is defined to be a predicate on states that holds for all reachable states. A state is defined to be **reachable** if it can be reached from an initial state through transitions.

The following is a fairly established way for proving that a state predicate (i.e. a predicate on states) p_g (goal predicate) is an invariant (i.e. `true` for all reachable states).

> Find state predicates $p_1, \cdots, p_n$ ($n \in \{0, 1, \cdots\}$) that satisfies the following two conditions.
>
> (t) Let *init* be a state predicate that specifies the initial states (i.e. $init(s_i)$ `iff` (s_i is an initial state)), and s be any state, then ($init(s)$ `implies` (p_g `and` p_1 `and` $\cdots$ `and` p_n)(s)[5]) holds.
>
> (v) Let t be any transition, and s_t and s'_t be the current state and the next state of t, then ((p_g `and` p_1 `and` $\cdots$ `and` p_n)(s_t) `implies` (p_g `and` p_1 `and` $\cdots$ `and` p_n)(s'_t)) holds.

Conditions (t) and (v) are called an **initial state condition** and an **invariant condition**, and a state predicate like (p_g `and` p_1 `and` $\cdots$ `and` p_n) that satisfies these two conditions is called an **inductive invariant**.

[5] (p_g `and` p_1 `and` $\cdots$ `and` p_n)(s) $\stackrel{\text{def}}{=}$ ($p_g(s)$ `and` $p_1(s)$ `and` $\cdots$ `and` $p_n(s)$)

It is easly seen that an inductive invariant is an invariant, hence if conditions (t) and (v) are proved, (p_g `and` p_1 `and` $\cdots$ `and` p_n) is an invariant, and all of p_g, p_1, $\cdots$, p_n are invariants.

Notice that if p_g itself is an inductive invariant then $n = 0$. However, p_1, p_2, $\cdots$, p_n are almost always needed to be found for getting an inductive invariant, and to find them is an important and challenging part of the invariant verification. Moreover, we think that to describe an inductive invariant as a conjunction of independent and fundamental state predicates is a quite effective way to formalize the dynamic behaviors of a system under investigation.

In this section, several state predicates for QLOCK are defined for specifying initial states and an inductive invariant.

4.1 `PNAT+ac` and `STATEfuns`

The modules `PNAT+ac` and `STATEfuns` are used to define predicates on `State` in the following sections.

```
-- Peano Style Natural Numbers with _+_
mod! PNAT+ac {
[Nat]
op 0 : -> Nat {constr} .
op s_ : Nat -> Nat {constr} .
-- equality over the natural numbers
eq (0 = s(Y:Nat)) = false .
eq (s(X:Nat) = s(Y:Nat)) = (X = Y) .
-- associative and commutative _+_
op _+_ : Nat Nat -> Nat {assoc comm}
eq 0 + Y:Nat = Y .
eq (s X:Nat) + Y:Nat = s(X + Y) .
}
```

Notice that associativity and commutativity of `_+_` is declared, but it can be deduced from the two equations for `_+_`.

```
-- elementary functions on states
mod! STATEfuns {pr(PNAT+ac + STATE)
-- variable declarations
vars L1 L2 : Label . vars A1 A2 : Aid .
var S : State .  var Q : Qu .
-- the number of queues in a state
op #q : State -> Nat .
...
-- the number of a label in a state
op #ls : State Label -> Nat .
...
-- the number of an aid in a state
```

```
op #as : State Aid -> Nat .
...
-- the number of an aid in a queue
op #aq : Qu Aid -> Nat .
...
}
```

`pr(PNAT+ac + STATE)` is same as "`pr(PNAT+ac) pr(STATE)`". "`...`" indicates omission. A variable declaration starts with `var`, contains variable and its sort before and after ":", and ends with ".". `vars` is plural of `var` and makes it possible to declare many varialbes of same sort together.

4.2 PNAMEcj, STATEpred1, and INIT

The predicates needed to define well formed states and initial states are defined using functions defined in `PNAT+ac` and `STATEfuns`. Notice that a state (i.e. a ground[6] term of sort `State`) is *well formed* if it contains (1) exactly one queue observer, (2) at least one agent observer, and (3) for any agent id $a1$ of sort `Aid` at most one agent observer of the form (`lb[`$a1$`]: L:Label`).

```
-- names of predicates on states and conjunction of the predicates
mod! PNAMEcj {pr(STATE)
-- names of predicates on States and sequences of them
[Pname < PnameSeq]
op (_ _) : PnameSeq PnameSeq -> PnameSeq {assoc} .
-- conjunction of predicates indicated in PnameSeq
op cj : PnameSeq State -> Bool .
eq cj(PN:Pname PNS:PnameSeq,S:State) = cj(PN,S) and cj(PNS,S) .
}
-- predicates on states for well formed states and intitial states
mod! STATEpred1 {pr(STATEfuns) ex(PNAMEcj)
-- one queue in a state
op 1q : -> Pname .
eq[1q]: cj(1q,S:State) = (#q(S) = (s 0)) .
-- no duplication of an Aid in a state
op 1a : -> Pname .
...
-- qas pattern, only the state with this pattern is well formed
pred qas : State .
eq qas((qu: Q:Qu)(lb[A:Aid]: L:Label) S:State) = true .
op qas : -> Pname .
eq[qas]: cj(qas,S:State) = qas(S) .
-- well formed states
op wfs : -> Pname . eq wfs = qas 1q 1a .
```

[6] a ground term is a term without variables.

```
-- there is exactly one empty queue
op qe : -> Pname .
...
-- any Aid is in rs status, i.e. no ws, no cs
op allRs : -> Pname .
...
}
-- an initial state predicate
mod! INIT {pr(STATEpred1)
op init : -> PnameSeq . eq init = wfs qe allRs .
-- initial state predicate
pred init : State . eq init(S:State) = cj(init,S) .
}
```

`ex(_)` indicates an **extending** importation, and declares to import a module without changing equality between already exit elements but introducing new elements of already exist sorts.

Notice that because of the module `PNAMEcj`, the conjunction of predicates can be defined by an equation like "`eq init = wfs qe allRs .`".

4.3 STATEpred2 and INV

In the following two modules `STATEpred2` and `INV`, an inductive invariant of QLOCK is developed as a conjunction of six predicates `wfs`, `mx`, `qep`, `rs`, `ws`, and `cs`. Notice that at this moment it is not known whether it is an inductive invariant; section 5.2 gives a proof score for proving that it is an inductive invariant.

Notice also that the development of an inductive invariant is inherently interactive activity involving proof score constructions and specification modifications, and there is no generally effective way for it. However, we think that describing state predicates (e.g. `qep`, `rs`, `ws`, `cs`) for characterising all cases that are indicated by the state configurations (e.g. `if queue is empty`, `if agent is in rs`, `if agent is in ws`, `if agent is in cs`) has a good chance to evolve into an inductive invariant.

```
-- predicates on states for an inductive invariant predicate
mod! STATEpred2 {pr(STATEpred1)
-- variable declarations
var L : Label . var A : Aid .
var S : State . var Q : Qu .
-- mutual exclusion property: at most one agent is with cs
-- this is the goal predicate
op mx : -> Pname .
eq[mx]: cj(mx,S) = ((#ls(S,cs) = 0) or (#ls(S,cs) = (s 0))) .
-- several fragment predicates for an inductive invariant
ops qep rs ws cs : -> Pname .
```

```
-- if queue is empty
eq[qep]: cj(qep,((qu: Q)(lb[A]: L) S))
         = ((Q = empQ) implies
            (#ls(((lb[A]: L) S),cs) = 0)) .
-- if agent is in rs
eq[rs]: cj(rs,((qu: Q)(lb[A]: L) S))
        = ((L = rs) implies (#aq(Q,A) = 0)) .
-- if agent is in ws
eq[ws]: cj(ws,((qu: Q)(lb[A]: L) S))
        = ((L = ws) implies
           ((#aq(Q,A) = (s 0)) and
            ((A = hd(Q)) implies (#ls(S,cs) = 0)))) .
-- if agent is in cs
eq[cs]: cj(cs,((qu: Q)(lb[A]: L) S))
        = ((L = cs) implies ((A = hd(Q)) and
                             (#aq(tl(Q),A) = 0)and
                             (#ls(S,cs) = 0))) .
}
-- an inductive invariant predicate
mod! INV {pr(STATEpred2)
op inv : -> PnameSeq .
eq inv = wfs mx qep rs ws cs .
pred inv : State .
eq inv(S:State) = cj(inv,S) .
}
-- property specification of QLOCK
mod! QLOCKprop{pr(INIT + INV)}
```

5 Proof Scores for Generate&Check Methods

As explained in Section 4, proving (t) initial state condition and (v) invariant condition is sufficient for proving that the goal predicate p_g is an invariant. For the QLOCK specification given in Sections 3 and 4, the goal predicate is `mx` and the two conditions are given as follows.

(t) Let s be any state (i.e. ground state term or ground term of sort `State`), then (`init`(s) `implies` `inv`(s)) holds.
(v) Let t be any transition defined by the `trans` rules `wt`, `ty`, `ex`, and cnt(t) and nxt(t) be the current state and the next state of t, then (`inv`(cnt(t)) `implies` `inv`(nxt(t)) holds.

5.1 Proof score for the initial state condition

If the reduction command in the following CafeOBJ code returns `true`, it means that the initial state condition for QLOCK has been proved by using all equations in the module `QLOCKprop` as rewriting rules from left to right.

```
open QLOCKprop .
op s : -> State .
red init(s) implies inv(s) .
close
```

Notice that the fresh constant s acts as a variable in the reduction[7]. Unfortunately, the reduction does not return `true`, and we need to do case analysis about the state s.

The idea of the generate&check methods is as follows.

(g) Generate finite number of state terms (with or without variables) sp_1, sp_2, $\cdots$, sp_n that subsume all possible infinite states (i.e. ground state terms).
(c) Check that the state predicate to be proved holds for all the finite state terms sp_1, sp_2, $\cdots$, sp_n.

For QLOCK's initial state condition, the state predicate to be proved is (`init-c(S:State)` $\stackrel{\text{def}}{=}$ `init(S) implies inv(S)`). Because (`init`(s) = `false`) for any ground state term s that is not an instance of the state term `((qu: Q:Qu) (lb[A:Aid] L:Label) S:State)` that subsume all the well formed ground state terms, the following Method 3 proves the initial state condition.

For describing the Method 3 precisely we need formal definitions of instance and cover. The definition of "instance" is established common one, but the definition of "cover" is unique even though there are several similar definitions.

Let $T(X)$ denote the set of terms with variables X. An **assignment** $a : X \rightarrow T(X)$ assigns terms in $T(X)$ to the variables X, and it can be naturally extended to $a : T(X) \rightarrow T(X)$. For a term $s \in T(X)$, $a(s)$ represents the term obtained by replacing each variable in s by the assigned term[8].

Definition 1 [**Instance**] The term $si \in T(X)$ is defined to be an **instance** of a term $s \in T(X)$ iff there exits an assignment $a : X \rightarrow T(X)$ such that $si = a(s)$. $\square$

Definition 2 [**Cover**] Let C and S be subsets of $T(X)$. C is defined to **cover** S iff for any ground instance sgi of any $s \in S$, there exits $si \in C$ such that sgi is an instance of si and si is an instance of s. $\square$

Method 3 [**Generate&Check-Init**][9]

(g) Generate state terms $sp_1, \cdots, sp_n$ that cover the state term (the term `((qu: Q:Qu)(lb[A:Aid]: L:Label) S:State)` for QLOCK) that subsumes all the well formed states (ground state terms)[10].
(c) Check that `init-c`(sp_i) reduces to `true` for any $i \in \{1, \cdots, n\}$. $\square$

---

[7] It is a well know fact called "theorem of constants"[8].
[8] The definition here does not treat order-sorted signature explicitly and is rather casual. More proper and formal definition can be found in [7].
[9] Correctness of this method is proved in Proposition 6.
[10] Notice that ((A covers B) and (B subsumes C)) implies (A subsumes C).

By recognizing that the state predicate `inv` are defined using the predicates like `(Q = empQ)`, `(A = hd(Q))`, `(L = rs)`, `(L = ws)`, `(L = cs)` (Section 4.3), it is natural to try to generate the covering state terms based on the following case analyses of the state term `((qu: Q:Qu)(lb[A:Aid]: L:Label) S:State)`. (1) whether `Q = empQ` or `Q = (b1 & q)`. (2) whether `A = b1` or `A = b2`. (3) which of `(L = rs)`, `(L = ws)`, or `(L = cs)` holds. Where `b1`, `b2` are constant literals and `q`, `s` are constants for representing arbitrary objects of specific sorts.

By representing the state term `((qu: Q:Qu)(lb[A:Aid]: L:Label) S:State)` as the sequence of arguments `(Q:Qu,A:Aid,L:Label,S:State)`, the covering state terms generated by the above case anayses can be defined as

```
[(empQ;(b1 & q)),(b1;b2),(rs;ws;cs),(s)]
```

that is expanded into the following 12 sequences of arguments. Notice that "`;`" enumerate all possible options, and constant literals and constants are used instead of variable literals and variables thanks to "theorem of conatants".

```
[empQ,b1,rs,s] || [empQ,b1,ws,s] || [empQ,b1,cs,s] ||
[empQ,b2,rs,s] || [empQ,b2,ws,s] || [empQ,b2,cs,s] ||
[(b1 & q),b1,rs,s] || [(b1 & q),b1,ws,s] || [(b1 & q),b1,cs,s] ||
[(b1 & q),b2,rs,s] || [(b1 & q),b2,ws,s] || [(b1 & q),b2,cs,s]
```

Notice the followings. (1) the current fairly simple "expansion algorithm" generate the second line, but the first line and second line need not be distinguished, and the second line is redundant. (2) the third line represents the cases in which the top of the queue and the agent id is same, and the fourth line represent the cases in which the two are different. (3) the generated state terms (i.e. the sequences of arguments) covers the state term `((qu: Q:Qu)(lb[A:Aid]: L:Label) S:State)`.

By defining

```
eq v(Q:Qu,A:Aid,L:Label,S:State)
   = init((qu: Q)(lb[A]: L) S) implies inv((qu: Q)(lb[A]: L) S) .
```

and checking that `v(Q,A,L,S)` reduces to `true` for any of the generated 12 argument sequences (i.e. state terms), the proof of the initial state condition of QLOCK is completed. The following is a proof score fragment for executing the proof explained above. When executed, the last reduction command returns "`($):Ind`" and shows that the predicate `v(Q,A,L,S)` reduces to `true` for all the generated argument sequences.[11]

```
-- generate and check all possible cases for the initial state condition
mod! CKallCasesInit {ex(GENcases(QLOCKinit))
-- Aid constant literals
```

[11] Readers are recommended to execute the CafeOBJ code in the ANNEX with the CafeOBJ system (http://www.ldl.jaist.ac.jp/cafeobj/download.html) and check that all reductions return expected results.

```
[AidConLt < Aid]
eq (B1:AidConLt = B2:AidConLt) = (B1 == B2) .
ops b1 b2 : -> AidConLt .
-- arbitray constants
op q : -> Qu . op s : -> State .
-- function for generating and checking all possible
-- states of the pattern:
--  ((qu: Q:Qu)(lb[A:Aid]: L:Label) S:State)
op gen&ck : -> IndTr .
-- a term of sort IndTr for checking all possible cases
eq gen&ck = ($ | mmi[(empQ;(b1 & q)),(b1;b2),(rs;ws;cs),(s)]) .
pr(FACTtbu)
}
-- reduction for verification of initial state condition
red in CKallCasesInit : gen&ck .
```

Notice that (1) `v(Q,A,L,S)` is defined in `QLOCKinit`, (2) `mmi` is defined in `GENcases` and generates the covering argument sequences (i.e. covering state terms) by expanding options specified by "_;_", (3) `v(Q,A,L,S)` is checked to reduce to `true` for all the cases by reducing `gen&ck`, (4) `FACTtbu` contains the following 2 theorems (that can be proved easily) about basic data types `Nat` and `Qu` that are needed for making the checks successful.

```
 eq ((M:Nat + N:Nat) = 0) = ((M = 0) and (N = 0)) .
 eq #aq(Q:Qu & A1:Aid,A2:Aid) = if (A1 = A2) then (s 0) + #aq(Q,A2) else #aq(Q,A2) fi .
```

5.2 Proof score for the invariant condition

The proof score for the invariant condition is almost same as for the initial state condition except (1) all the infinite transitions should be subsumed instead of all the infinite states (i.e. ground state terms) and (2) the predicate to be checked for each generated case (i.e. covering term) is different.

In the generate&check methods, it is assumed that all transitions are defined by `trans` rules. As a matter of fact, for guaranteeing the correctness of the proof score, all the `trans` rules should be unconditional. We do not think this is a serous limitation, for almost always needed conditions can be incorporated into the right hand side of unconditional rules using built-in `if_then_else_fi` operator. Notice that all the QLOCK's `trans` rules `wt`, `ty`, `ex` are unconditional.

Let lhs(r) denote the left hand side of a `trans` rule r. Because any transition is defined by some `trans` rule, for any transition t_i there exits some `trans` rule r_i such that cnt(t_i) (the current state of t_i) is an instance of lhs(r_i). This fact suggests the possibility of subsuming all the infinite transitions by covering the set of left hand sides of all the finite `trans` rules.

Let us consider the following CafeOBJ code.

```
open (QLOCKprop + QLOCKsys) .
```

```
op inv-c : State State -> Bool . vars S SS : State .
eq inv-c(S,SS) =
   (not(S =(*,1)=>+ SS
        suchThat (not((inv(S) implies inv(SS)) == true)))) .
op s : -> State .
red inv-c(s,SS) .
close
```

Let "`op P : State State -> Bool .`" be a predicate, then the CafeOBJ's built-in search predicate:

$$\texttt{BSP(S:State,SS:State)} \stackrel{\mathrm{def}}{=} \texttt{(S =(*,1)=>+ SS suchThat P(S,SS))}$$

behaves as follows if `S` is given, and let `RESULT` be a Boolean indicator initially set to `false`. While {there is an untried pair of (`trans` rule r, matching m) such that `S` is an instance of $\mathrm{lhs}(r)$}[12] do the following {do the one step transition with (r, m), bind the obtained next state to `SS`, and if `P(S,SS)` reduces to `true` then bind `true` to `RESULT`}. `BSP(S,SS)` reduces to `true` if `RESULT` is `true` and to `false` otherwise.

Hence, if `inv-c(s,SS)` in the above CafeOBJ code reduces to `true`, it implies that there is no transition $\hat{t}$ from the state `s` such that (`inv`($\mathrm{cnt}(\hat{t})$) `implies` `inv`($\mathrm{nxt}(\hat{t})$)) does not reduces to `true`. In other words, any transitions from `s` preserves the predicate `inv`, that is for any transition t from `s` (`inv`($\mathrm{cnt}(t)$) `implies` `inv`($\mathrm{nxt}(t)$)) reduces to `true`.

Based on above arguments, if we can check that `inv-c(s,SS)` reduces to `true` for all the states (i.e. all the ground state terms), the invariant condition is proved. As a matter of fact, because there is no transitions from the state that is not an instance of the left hand side of any `trans` rule, we can only consider the state that is an instance of the left hand side of some `trans` rule.

As a result, the following Method 4 proves the invariant condition.

Method 4 [Generate&Check-Inv][13]

(g) Generate state terms $sp_1, \cdots, sp_n$ that cover the set of left hand sides of all the `trans` rules (i.e. {`wt`, `ty`, `ex`} for QLOCK).
(c) Check that `inv-c(`sp_i`,SS:State)` reduces to `true` for any $i \in \{1, \cdots, n\}$. $\square$

The left hand sides of the three `trans` rules `wt`, `ty`, `ex` are

```
((qu: Q:Qu)(lb[A:Aid]: rs) S:State),
((qu: (A:Aid & Q:Qu))(lb[A]: ws) S:State),
((qu: (A1:Aid & Q:Qu))(lb[A2:Aid]: cs) S:State)
```

and these three state terms can be represented by the three argument sequences

[12] Notice that a single `trans` rule defines two or more transitions from a state with different matchings. Notice also that a matching determines an assignment.

[13] Correctness of this method is proved in Proposition 7.

```
(Q:Qu,A:Aid,rs,S:State),
((A:Aid & Q:Qu),A,ws,S:State),
((A1:Aid & Q:Qu),A2:Aid,cs,S:State)
```

For checking instances of these argument sequences, `inv-c(s,SS)` is specified as follows in the module `QLOCKinv`.

```
eq v(Q:Qu,A:Aid,L:Label,S:State,SS:State) =
   (not(((qu: Q) (lb[A]: L) S) =(*,1)=>+ SS suchThat
        (not((inv((qu: Q)(lb[A]: L) S) implies inv(SS)) == true)))) .
```

It is seen that
{ `[(empQ;(b1 & q)),(b1;b2),(rs;ws;cs),(s),(SS)]` }
covers

```
{ [(empQ;(b1 & q)),(b1;b2),(rs),(s),(SS)],
  [(b1 & q),(b1),(ws),(s),(SS)],
  [(b1 & q),(b1;b2),(cs),(s),(SS)] }
```

and this in turn covers

```
{ (Q:Qu,A:Aid,rs,S:State),
  ((A:Aid & Q:Qu),A,ws,S:State),
  ((A1:Aid & Q:Qu),A2:Aid,cs,S:State) }.
```

Since _covers_ is a transitive relation (i.e. ((s_a covers s_b) and (s_b covers s_c)) implies (s_a covers s_c)), the above implies that the first covers the third. Notice that constant literals and constants are used for variable literals and variables depending on the contexts.

By the above arguments the following main module of the proof score for the invariant condition is obtained, and the reduction in the last line returns "`($):Ind`" as expected. Hence, the invariant condition is proved.

```
-- a module to generate and check all possible transitions
mod! CKallCasesInv {ex(GENcases(QLOCKinv))
-- Aid constant literals
[AidConLt < Aid]
eq (B1:AidConLt = B2:AidConLt) = (B1 == B2) .
ops b1 b2 : -> AidConLt .
-- constants declarations
op q : -> Qu . op s : -> State .
-- function for generating and checking all possible
-- transitions defined by the module WT, TY, EX
op gen&ck : State -> IndTr .
var SS : State .
eq gen&ck(SS) =
   ($ | mmi[(empQ;(b1 & q)),(b1;b2),(rs;ws;cs),(s),(SS)]) .
```

```
pr(FACTtbu)
}
-- reduction for verification of invariant condition
red in CKallCasesInv : gen&ck(SS) .
```

5.3 Correctness of the generate&check methods

This section proves correctness of the the generate&check methods described in Sections 5.1 and 5.2.

For $s_s, s_t \in T(X)$, let $(s_s \xrightarrow{*} s_t)$ denote that there exits a rewriting sequence of length $n \geq 0$ from s_s to s_t by using all the equations available as rewriting rules from left to right. Let also $(s_s \rightarrow s_t)$ denote that there exits a one step rewriting.

The following lemma shows a most important property of the cover sets.

Lemma 5 [Cover Lemma] Let $C, S \subseteq T(X)$, $C \stackrel{\text{def}}{=} \{c_1, c_2, \cdots, c_m\}$, $S \stackrel{\text{def}}{=} \{s_1, s_2, \cdots, s_n\}$, and p be a predicate. If (C covers S) and $(p(c_i) \xrightarrow{*} \texttt{true})$ for all $i \in \{1, 2, \cdots, m\}$, then for any $j \in \{1, 2, \cdots, n\}$, for any ground instance sgi_j of s_j, $(p(sgi_j) \xrightarrow{*} \texttt{true})$.

(proof) Because (C covers S), there exits $c_k \in C$ and an assignment $(a : X \rightarrow T(X))$ such that $sgi_j = a(c_k)$. For any two terms $u, u' \in T(X)$, $(u \rightarrow u')$ implies $(a(u) \rightarrow a(u'))$. Therefore, the assumed rewriting sequence $(p(c_k) \xrightarrow{*} \texttt{true})$ can be executed literally on sgi_j and we get $(p(sgi_j) \xrightarrow{*} \texttt{true})$. □

Proposition 6 [Generate&Check-Init] If the (g) and (c) of the Method 3 are achieved successively, then `init-c`(s) holds for any state (ground state term) s.

(proof) If a state s_i is not an instance of `((qu: Q:Qu)(lb[A:Aid]: L:Label) S:State)`, `init`(s_i) does not holds by definition, hence `init-c`(s_i) holds. If a state s_w is an instance of `((qu: Q:Qu)(lb[A:Aid]: L:Label) S:State)`, then by (g), (c), Lemma 5, and the fact ((`init-c`(sp_i) reduces to `true`) implies (`init-c`(sp_i) $\xrightarrow{*}$ `true`)), we get (`init-c`(s_w) $\xrightarrow{*}$ `true`)). Hence `init-c`(s_w) holds. □

Proposition 7 [Generate&Check-Inv] If the (g) and (c) of the Method 4 are achieved successively, then (`inv`(cnt(t)) `implies` `inv`(nxt(t))) holds for any transition t.

(proof) Since it is assumed that all transitions are defined by unconditional `trans` rules, cnt(t) should be an instance of the left hand sides of some `trans` rule for any transition t. Therefore, we can check all the transitions by checking all the states (i.e. ground state terms) that are instances of the left hand sides of the `trans` rules. Let $(s \stackrel{r_k, a_k}{\Rightarrow} s')$ denote that there is a transition from a state term s to a state term s' with a `trans` rule r_k and an assignment a_k. Let rgi be any ground state term that is an instance of the left hand side of some `trans` rule. Any transition from the state rgi can be represented as $(rgi \stackrel{r, a_1}{\Rightarrow} rgi')$ for some `trans` rule r, some assignment a_1, and some ground state term rgi'. Notice that $rgi = a_1(\text{lhs}(r))$.

Because of (g) there exit sp_j ($j \in \{1, 2, \cdots, n\}$) and two assignments a_2 and a_3 such that ($rgi = a_2(sp_j)$) and ($sp_j = a_3(\mathrm{lhs}(r))$). Since ($sp_j = a_3(\mathrm{lhs}(r))$) and r is unconditional, we get ($sp_j \stackrel{r,a_3}{\Rightarrow} sp'_j$) for some state term sp'_j, such that ($rgi = a_2(sp_j) = a_2(a_3(\mathrm{lhs}(r))) = a_1(\mathrm{lhs}(r))$) (i.e. $a_1(\_) = a_2(a_3(\_))$) and ($rgi' = a_2(sp'_j)$). Hence, if ($rgi \stackrel{r,a}{\Rightarrow} rgi'$) then ($sp_h \stackrel{r,\hat{a}}{\Rightarrow} sp'_h$) for some $h \in \{1, 2, \cdots, n\}$ and some assignment $\hat{a}$. Moreover, (`inv-c`(sp_h,`SS:State`) $\stackrel{*}{\rightarrow}$ `true`) because of (c), there exits an assignment $\tilde{a}$ such that (($rgi = \tilde{a}(sp_h)$) and ($rgi' = \tilde{a}(sp'_h)$)), and for any two terms u, u' (($u \rightarrow u'$) implies ($\tilde{a}(u) \rightarrow \tilde{a}(u')$)). Therefore, we get (`inv-c`(rgi,`SS:State`) $\stackrel{*}{\rightarrow}$ `true`), and (`inv`(cnt(t)) `implies` `inv`(nxt(t))) holds for any transition t from rgi. □

Notice that the condition "$si \in C$ is an instance of $s \in S$" in Definition 2 is necessary in Proposition 7, but not in Lemma 5 and Proposition 6.

6 Conclusions

The proposed generate&check methods for invariant verification of transition systems are summarized as follows.

1. Model and specify a problem/system with OTS (observational transition system) in which states are represented as sets of observers and transitions are specified with unconditional `trans` rules.
2. (g) Generate state terms sp_1, sp_2, $\cdots$, sp_m that subsume all the well formed ground state terms such that (c) `init-c`(sp_i) reduces to `true` for any $i \in \{1, 2, \cdots, m\}$.
3. (g) Generate state terms sp_1, sp_2, $\cdots$, sp_n that cover the set of left hand sides of all the `trans` rules such that (c) `inv-c(`sp_i`,SS:State)` reduces to `true` for any $i \in \{1, 2, \cdots, n\}$.

We have shown that the methods proposed are nicely coded and executed in the CafeOBJ language/system using the QLOCK example. However, the methods and theory presented are not specific to QLOCK, but are general enough to be applied to any specification that satisfies stated assumptions.

There are quite a few researches on search techniques in model checking [3, 9]. It is interesting to observe that what we have done in Method 4 is a search in state space across all one step transitions, whereas the search for model checking is along time axis (i.e. transition sequences) as shown in Figure 2.

This paper only shows CafeOBJ specification and proof score for the rather small QLOCK example. We have, however, already checked that the methods proposed are effective for more larger example like ABP (Alternating Bit Protocol [14]). As a matter of fact, "generate&check" methods should be more important for large problems, for it is difficult to do case analyses manually for them. Once state configurations are properly designed, large number of cases (i.e. elements of cover set) are generated and checked easily, and it is an important future issue to construct proof scores for important problems/systems of significant sizes and do experiments for learning efficient way to obtain a cover set that has high possibility of being checked successfully.

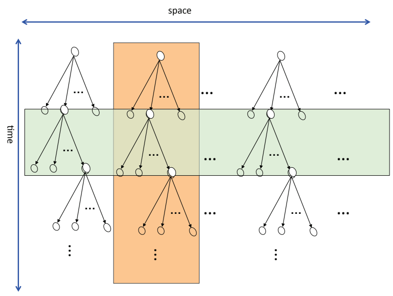

Fig. 2. Searches on Time versus Space

References

1. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions. Springer (2004)
2. CafeOBJ: Web page. http://www.ldl.jaist.ac.jp/cafeobj/ (2013)
3. Clarke, E.M., Grumberg, O., Peled, D.: Model checking. MIT Press (2001)
4. Dong, J.S., Zhu, H. (eds.): Formal Methods and Software Engineering - 12th International Conference on Formal Engineering Methods, ICFEM 2010, Shanghai, China, November 17-19, 2010. Proceedings, Lecture Notes in Computer Science, vol. 6447. Springer (2010)
5. Futatsugi, K.: Verifying specifications with proof scores in CafeOBJ. In: Proc. of 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006). pp. 3–10. IEEE Computer Society (2006)
6. Futatsugi, K.: Fostering proof scores in CafeOBJ. In: Dong and Zhu [4], pp. 1–20
7. Futatsugi, K., Gâinâ, D., Ogata, K.: Principles of proof scores in CafeOBJ. Theor. Comput. Sci. 464, 90–112 (2012)
8. Goguen, J.: Theorem Proving and Algebra. [Unpublished Book] (now being planned to be up on the web for the free use)
9. Grumberg, O., Veith, H. (eds.): 25 Years of Model Checking - History, Achievements, Perspectives, Lecture Notes in Computer Science, vol. 5000. Springer (2008)
10. Guttag, J.V., Horning, J.J., Garland, S.J., Jones, K.D., Modet, A., Wing, J.M.: Larch: Languages and Tools for Formal Specification. Springer (1993)
11. Meseguer, J.: Twenty years of rewriting logic. J. Log. Algebr. Program. 81(7-8), 721–781 (2012)
12. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
13. Ogata, K., Futatsugi, K.: Proof scores in the OTS/CafeOBJ method. In: Najm, E., Nestmann, U., Stevens, P. (eds.) FMOODS. Lecture Notes in Computer Science, vol. 2884, pp. 170–184. Springer (2003)
14. Ogata, K., Futatsugi, K.: Simulation-based verification for invariant properties in the OTS/CafeOBJ method. Electr. Notes Theor. Comput. Sci. 201, 127–154 (2008)
15. Ogata, K., Futatsugi, K.: A combination of forward and backward reachability analysis methods. In: Dong and Zhu [4], pp. 501–517

ANNEX

```
** ==============================================================
** ======= Specification and Proof Score for QLOCK ===============
** ==============================================================

** ==============================================================
** ================ System Specification ========================
** ==============================================================

-- three labels for indicating status of each agent
mod! LABEL {
-- label literals and labels
[LabelLt < Label]
-- rs: remainder section
-- ws: waiting section
-- cs: critical section
ops rs ws cs : -> LabelLt {constr} .
eq (L1:LabelLt = L2:LabelLt) = (L1 == L2) .
}

-- agent identifiers
mod* AID {[Aid]}

-- ==============================================================
-- queue (first in first out storage)
mod! QUEUE (X :: TRIV) {
-- elements and their queues, Elt comes from (X :: TRIV)
[Elt.X < Qu]
-- error elements and error queues
[Elt.X < Elt&Err]  [Qu < Qu&Err]
-- empty queue
op empQ : -> Qu {constr} .
-- assoicative queue constructors with id: empQ
op (_&_) : Qu Qu -> Qu {constr assoc id: empQ} .
op (_&_) : Qu&Err Qu&Err -> Qu&Err {constr assoc id: empQ} .
-- equality _=_ over Qu&Err
eq (empQ = (E:Elt & Q:Qu&Err)) = false .
ceq ((E1:Elt & Q1:Qu&Err) = (E2:Elt & Q2:Qu&Err))
    = ((E1 = E2) and (Q1 = Q2))
    if not((Q1 = empQ) and (Q2 = empQ)) .
-- head
op hd_ : Qu&Err -> Elt&Err .
eq hd(E:Elt & Q:Qu&Err) = E .
-- hd(empQ) is not defined intentionally, an error handling method
```

```
-- tail
op tl_ : Qu&Err -> Qu&Err .
eq tl(E:Elt & Q:Qu&Err) = Q .
-- tl(empQ) is not defined intentionally, an error handling method
}

-- ================================================================
-- observers
mod! OBS {
pr(LABEL)
pr(QUEUE(AID{sort Elt -> Aid}))
-- there are two kinds of obserbers
[Obs]
op (qu:_ ) : Qu -> Obs {constr} .
op (lb[_]:_) : Aid Label -> Obs {constr} .
}

-- generic set
mod! SET(X :: TRIV) {
[Elt.X < Set]
-- empty set
op empty : -> Set {constr} .
-- assicative and commutative set constructor with identity empty
op (_ _) : Set Set -> Set {constr assoc comm id: empty} .
-- (_ _) is idempotent
eq E:Elt E = E .
}

-- a state is defined as a set of observers
mod! STATE {pr(SET(OBS{sort Elt -> Obs})*{sort Set -> State})}

-- ================================================================
-- wt: want transition
mod! WT {pr(STATE)
trans[wt]:
     ((qu: Q:Qu)(lb[A:Aid]: rs) S:State)
     => ((qu: (Q & A))(lb[A]: ws) S) .
}
-- ty: try transition
mod! TY {pr(STATE)
trans[ty]:
     ((qu: (A:Aid & Q:Qu))(lb[A]: ws) S:State)
     => ((qu: (A & Q))(lb[A]: cs) S) .
}
-- ex: exit transition
```

```
mod! EX {pr(STATE)
trans[ex]:
   ((qu: (A1:Aid & Q:Qu))(lb[A2:Aid]: cs) S:State)
   => ((qu: Q)(lb[A2]: rs) S) .
}

-- ================================================================
-- system specification of QLOCK
mod! QLOCKsys{pr(WT + TY + EX)}

** ================================================================
** ================ Property Specification ======================
** ================================================================

-- ================================================================
-- for defining state predicates

-- Peano Style Natural Numbers with ac-_+_
mod! PNAT+ac {
  [Nat]
  op 0 : -> Nat {constr} .
  op s_ : Nat -> Nat {constr} .
  -- equality over the natural numbers
  eq (0 = s(Y:Nat)) = false .
  eq (s(X:Nat) = s(Y:Nat)) = (X = Y) .
  -- associative and commutative _+_
  op _+_ : Nat Nat -> Nat {assoc comm}
  eq 0 + Y:Nat = Y .
  eq (s X:Nat) + Y:Nat = s(X + Y) .
}

-- elementary functions on states
mod! STATEfuns {pr(PNAT+ac + STATE)
-- variable declarations
vars L1 L2 : Label . vars A1 A2 : Aid .
var S : State .  var Q : Qu .
-- the number of queues in a state
op #q : State -> Nat .
eq #q(empty) = 0 .
eq #q((qu: Q) S) = (s 0) + #q(S) .
eq #q((lb[A1]: L1) S) = #q(S) .
-- the number of a label in a state
op #ls : State Label -> Nat .
eq #ls(empty,L1) = 0 .
eq #ls(((qu: Q) S),L1) = #ls(S,L1) .
```

```
eq #ls(((lb[A1]: L1) S),L2) =
   if (L1 = L2) then (s 0) + #ls(S,L2)
   else #ls(S,L2) fi .
-- the number of an aid in a state
op #as : State Aid -> Nat .
eq #as(empty,A1) = 0 .
eq #as((qu: Q) S,A1) = #as(S,A1) .
eq #as(((lb[A1]: L1) S),A2) =
   if (A1 = A2) then (s 0) + #as(S,A2)
   else #as(S,A2) fi .
-- the number of an aid in a queue
op #aq : Qu Aid -> Nat .
eq #aq(empQ,A1) = 0 .
eq #aq(A1 & Q,A2) =
   if (A1 = A2) then (s 0) + #aq(Q,A2)
   else #aq(Q,A2) fi .
}

-- names of predicates on states and conjunction of the predicates
mod! PNAMEcj {pr(STATE)
-- names of predicates on States and sequences of them
[Pname < PnameSeq]
op (_ _) : PnameSeq PnameSeq -> PnameSeq {assoc} .
-- conjunction of predicates indicated in PnameSeq
op cj : PnameSeq State -> Bool .
eq cj(PN:Pname PNS:PnameSeq,S:State)
   = cj(PN,S) and cj(PNS,S) .
}

-- ================================================================
-- predicates on states for well formed states and intitial states
mod! STATEpred1 {pr(STATEfuns)ex(PNAMEcj)
-- one queue in a state
op 1q : -> Pname .
eq[1q]: cj(1q,S:State) = (#q(S) = (s 0)) .
-- no duplication of an Aid in a state
op 1a : -> Pname .
eq[1a]: cj(1a,empty) = true .
eq[1a]: cj(1a,((lb[A:Aid]: L:Label) S:State)) =
        (#as(S,A) = 0) and cj(1a,S) .
eq[1a]: cj(1a,((qu: Q:Qu) S:State)) = cj(1a,S) .
-- qas pattern, only the state with this pattern is well formed
pred qas : State .
eq qas((qu: Q:Qu)(lb[A:Aid]: L:Label) S:State) = true .
op qas : -> Pname .
```

```
eq[qas]: cj(qas,S:State) = qas(S) .
-- well formed states
op wfs : -> Pname .
eq wfs = qas 1q 1a .
-- there is exactly one empty queue
op qe : -> Pname .
eq[qe]: cj(qe,empty) = false .
eq[qe]: cj(qe,((lb[A:Aid]: L:Label) S:State))
        = cj(qe,S) .
eq[qe]: cj(qe,((qu: Q:Qu) S:State))
        = (Q = empQ) and (#q(S) = 0) .
-- any Aid is in rs status, i.e. no ws, no cs
op allRs : -> Pname .
eq[allRs]: cj(allRs,S:State) = (#ls(S,ws)= 0) and (#ls(S,cs)= 0) .
}

-- ==================================================================
-- an initial state predicate
mod! INIT {pr(STATEpred1)
op init : -> PnameSeq .
eq init = wfs qe allRs .
-- initial state predicate
pred init : State .
eq init(S:State) = cj(init,S) .
}

-- ==================================================================
-- predicates on states for an inductive invariant predicate
mod! STATEpred2 {pr(STATEpred1)
-- variable declarations
var L : Label . var A : Aid .
var S : State . var Q : Qu .
-- mutual exclusion property: at most one agent is with cs
-- this is the goal predicate
op mx : -> Pname .
eq[mx]: cj(mx,S) = ((#ls(S,cs) = 0) or (#ls(S,cs) = (s 0))) .
-- several fragment predicates for an inductive invariant
ops qep rs ws cs : -> Pname .
-- if queue is empty
eq[qep]: cj(qep,((qu: Q)(lb[A]: L) S))
         = ((Q = empQ) implies
            (#ls(((lb[A]: L) S),cs) = 0)) .
-- if agent is in rs
eq[rs]: cj(rs,((qu: Q)(lb[A]: L) S))
        = ((L = rs) implies (#aq(Q,A) = 0)) .
```

```
-- if agent is in ws
eq[ws]: cj(ws,((qu: Q)(lb[A]: L) S))
        = ((L = ws) implies
           ((#aq(Q,A) = (s 0)) and
            ((A = hd(Q)) implies (#ls(S,cs) = 0)))) .
-- if agent is in cs
eq[cs]: cj(cs,((qu: Q)(lb[A]: L) S))
        = ((L = cs) implies ((A = hd(Q)) and
                             (#aq(tl(Q),A) = 0)and
                             (#ls(S,cs) = 0))) .
}

-- ================================================================
-- an inductive invariant predicate
mod! INV {pr(STATEpred2)
op inv : -> PnameSeq .
eq inv = wfs mx qep rs ws cs .
pred inv : State .
eq inv(S:State) = cj(inv,S) .
}

-- ================================================================
-- property specification of QLOCK
mod! QLOCKprop{pr(INIT + INV)}

** ================================================================
** ================ Proof Score ==================================
** ================================================================

-- the following two modules describe the algorithm
-- for generating a finite set of patterns that cover
-- all possbile cases
-- by expanding alternatives indicated by (_;_)

-- predicate v that is to be checked
-- and indicator information constructor ii
mod* PREDtbC {
-- values and their sequences
[Val < ValSq]
op _,_ : ValSq ValSq -> ValSq {assoc} .
-- predicate to be checed
pred v_ : ValSq .
-- indicator information for analysis
[IndInfo]
op ii_ : ValSq -> IndInfo {constr} .
```

```
** v_ and ii_ shoud have a same arity
** as a sequence of 'Val's
}

-- generating a finit set of patterns
-- that cover all possible combinations
-- of values in a value sequence
mod! GENcases (X :: PREDtbC) {
-- sequences of values indicating
-- all possible alternatives
[Val < VlSq]
op _;_ : VlSq VlSq -> VlSq {assoc} .
-- sequence of ValSeq or VlSeq
[ValSq VlSq < SqSq]
op _,_ : SqSq SqSq -> SqSq {assoc} .
-- SqSq enclosures and their trees
[SqSqEn < SqSqTr]
op [_] : SqSq -> SqSqEn .
op _||_ : SqSqTr SqSqTr -> SqSqTr .

-- expanding alternatives indicated by (_;_)
-- into (_||_) as much as possible
var V : Val .
var VS : VlSq .
vars SS1 SS2 : SqSq .
eq [((V;VS),SS2)] = [(V,SS2)] || [(VS,SS2)] .
eq [(SS1,(V;VS),SS2)]
   = [(SS1,V,SS2)] || [(SS1,VS,SS2)] .
eq [(SS1,(V;VS))] = [(SS1,V)] || [(SS1,VS)] .

-- indicators and their trees
[Ind < IndTr]
op $ : -> Ind .
op _|_ : IndTr IndTr -> IndTr .
-- indicator constructor;
-- [IndInfo] comes from (X :: PREDtbC)
op i : Bool IndInfo -> Ind {constr} .
-- make indicator (mi) using
--  (v_ : ValSq -> Bool) and
--  (ii_ : ValSq -> IndInfo)
-- that come from (X :: PREDtbC)
op mi_ : ValSq -> Ind .
eq mi(VSQ:ValSq) = i(v(VSQ),ii(VSQ)) .

-- make make indicators (mmi):
```

```
-- translating a tree of SqSq (SqSqTr)
-- into a tree of indicators
op mmi_ : SqSqTr -> IndTr .
eq mmi(SST1:SqSqTr || SST2:SqSqTr)
   = (mmi SST1) | (mmi SST2) .
-- if all _;_ in SqSq disappear
-- then translate mmi to mi
eq mmi[VSQ:ValSq] = mi(VSQ) .

-- making all indicators with "true" disappear
eq i(true,II:IndInfo) | IT:IndTr = IT .
eq IT:IndTr | i(true,II:IndInfo) = IT .
}

-- ===============================================================
-- facts to be used
mod! FACTtbu {
pr(QLOCKprop)
-- necessary fact about _=_ on Nat
eq ((M:Nat + N:Nat) = 0) = ((M = 0) and (N = 0)) .

-- necessary fact about #aq
eq #aq(Q:Qu & A1:Aid,A2:Aid) = if (A1 = A2) then (s 0) + #aq(Q,A2)
                               else #aq(Q,A2) fi .
}

--> ==============================================================
--> Verification of the Initial State Condition:
-->  (for-all S:State)(init(S) implies inv(S))
--> ==============================================================
--> [0] cj(qas,s) = false .
open (INIT + QLOCKprop) .
op s : -> State .
eq cj(qas,s) = false .
red init(s) implies inv(s) .
close

--> [1] cj(qas,s) = true .
-- in this case, we can only consider a state
-- that is an instance of the pattern:
--  ((qu: Q:Qu)(lb[A:Aid]: L:Label) S:State)
--
-- define v_ and ii_ for checking
-- the initial state condition
mod! QLOCKinit {pr(INIT + QLOCKprop)
```

```
[Qu Aid Label State < Val < ValSq]
op _,_ : ValSq ValSq -> ValSq {assoc} .
-- predicate to be checked
op v_ : ValSq -> Bool .
eq v(Q:Qu,A:Aid,L:Label,S:State)
   = init((qu: Q)(lb[A]: L) S)
     implies inv((qu: Q)(lb[A]: L) S) .
[IndInfo]
op ii_ : ValSq -> IndInfo {constr} .
}
-- generate and check all possible cases
-- for the initial state condition
mod! CKallCasesInit {ex(GENcases(QLOCKinit))
-- Aid constant literals
[AidConLt < Aid]
eq (B1:AidConLt = B2:AidConLt) = (B1 == B2) .
ops b1 b2 : -> AidConLt .
-- arbitray constants
op q : -> Qu . op s : -> State .
-- function for generating and checking all possible
-- states of the pattern:
--  ((qu: Q:Qu)(lb[A:Aid]: L:Label) S:State)
op gen&ck : -> IndTr .
-- a term of sort IndTr for checking all possible cases
eq gen&ck = ($ | mmi[(empQ;(b1 & q)),
                     (b1;b2),
                     (rs;ws;cs),
                     (s)]) .
pr(FACTtbu)
}
-- reduction for verification of initial state condition
red in CKallCasesInit : gen&ck .

"{start of comment
-- ==================================================================
[(empQ;(b1 & q)),(b1;b2),(rs;ws;cs),(s)]

=is expanded to=>

[empQ,b1,rs,s] ||
[empQ,b1,ws,s] ||
[empQ,b1,cs,s] ||
[empQ,b2,rs,s] ||  -- redundant
[empQ,b2,ws,s] ||  -- redundant
[empQ,b2,cs,s] ||  -- redundant
```

```
[(b1 & q),b1,rs,s] ||
[(b1 & q),b1,ws,s] ||
[(b1 & q),b1,cs,s] ||
[(b1 & q),b2,rs,s] ||
[(b1 & q),b2,ws,s] ||
[(b1 & q),b2,cs,s]

-- ===============================================================
Hence,

{[(empQ;(b1 & q)),(b1;b2),(rs;ws;cs),(s)]}

covers

{((qu: Q:Qu)(lb[A:Aid]: L:Label) S:State)}

-- ===============================================================
end of comment}"


--> ==============================================================
--> Verification of the Invariant Condition:
--> (for-all (S->S'):State->State(One-Step-Transition))
-->                 (inv(S) implies inv(S'))
--> ==============================================================

--> [0] cj(qas,s) = false .
-- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
-- If 'cj(qas,s) = false' for any state 's', there is no
-- chance for the state 's' to match any of the three
-- transition rules wt, ty, or ex.  Hence, no transition
-- happens from the state 's' with 'cj(qas,s) = false',
-- and no need to consider this case.
-- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

--> [1] cj(qas,s) = true .
-- in this case, we can only consider a state
-- that is an instance of the pattern:
--  ((qu: Q:Qu)(lb[A:Aid]: L:Label) S:State)
--
-- define v_ and ii_: this module is an actual parameter
-- for the GENcases module
mod! QLOCKinv {pr(QLOCKsys + QLOCKprop)
-- val and ValSeq
[Qu Aid Label State < Val < ValSq]
```

```
op _,_ : ValSq ValSq -> ValSq {assoc} .
-- predicate to be checked
op v_ : ValSq -> Bool .
eq v(Q:Qu,A:Aid,L:Label,S:State,SS:State) =
    (not(((qu: Q) (lb[A]: L) S)  =(*,1)=>+ SS
         suchThat
         (not((inv((qu: Q)(lb[A]: L) S) implies inv(SS))
               == true)))) .
[IndInfo]
op ii_ : ValSq -> IndInfo {constr} .
}
-- a module to generate and check all possible transitions
mod! CKallCasesInv {ex(GENcases(QLOCKinv))
-- Aid constant literals
[AidConLt < Aid]
eq (B1:AidConLt = B2:AidConLt) = (B1 == B2) .
ops b1 b2 : -> AidConLt .
-- constants declarations
op q : -> Qu . op s : -> State .
--
-- function for generating and checking all possible
-- transitions defined by the module WT, TY, EX
op gen&ck : State -> IndTr .
-- variables to be bound by the built-in predicate:
--         (_=(*,1)=>+_suchThat_)
var SS : State .
eq gen&ck(SS) =
    ($ | mmi[(empQ;(b1 & q)),(b1;b2),(rs;ws;cs),(s),(SS)]) .
pr(FACTtbu)
}
-- reduction for verification of invariant condition
red in CKallCasesInv : gen&ck(SS) .

"{{start of comment
-- ================================================================
{[(empQ;(b1 & q)),(b1;b2),(rs;ws;cs),(s),(SS)]}

covers

{[(empQ;(b1 & q)),(b1;b2),(rs),(s),(SS)],
 [(b1 & q),(b1),(ws),(s),(SS)],
 [(b1 & q),(b1;b2),(cs),(s),(SS)]}

-- ================================================================
{[(empQ;(b1 & q)),(b1;b2),(rs),(s)(SS)]}
```

```
covers

{(((qu: empQ) (lb[A:Aid]: rs) S:State),(SS))
 (((qu: (A:Aid & Q:Qu)) (lb[A:Aid]: rs) S:State),(SS))}

covers

{WT:(((qu: Q:Qu) (lb[A:Aid]: rs) S:State),(SS))}

-- ==================================================================
{[(b1 & q),(b1),(ws),(s),(SS)]}

covers

{TY:(((qu: (A:Aid & Q:Qu)) (lb[A]: ws) S:State),(SS))}

-- ==================================================================
{[(b1 & q),(b1;b2),(cs),(s),(SS)]}

covers

{EX:(((qu: (A1:Aid & Q:Qu)) (lb[A2:Aid]: cs) S:State),(SS))}

-- ==================================================================
Hence,

{[(empQ;(b1 & q)),(b1;b2),(rs;ws;cs),(s),(SS)]}

covers

{WT:(((qu: Q:Qu) (lb[A:Aid]: rs) S:State),(SS)),
 TY:(((qu: (A:Aid & Q:Qu)) (lb[A]: ws) S:State),(SS)),
 EX:(((qu: (A1:Aid & Q:Qu)) (lb[A2:Aid]: cs) S:State),(SS))}

-- ==================================================================
end of comment}}"

eof
```