Generate & Check Method for Verifying Transition Systems 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

Abstract. An interactive theorem proving method for verification of infinite state transition systems is described.

The state space of a transition system is defined as a quotient set (i.e. a set of equivalence classes) of terms of a topmost sort `State`, and the transitions are defined with conditional rewrite rules over the quotient set. A property to be verified is either (1) an invariant (i.e. a state predicate that is valid for all reachable states) or (2) a (`p leads-to q`) property for two state predicates `p` and `q`. Where (`p leads-to q`) means that from any reachable state `s` with (`p(s) = true`) the system will get into a state `t` with (`q(t) = true`) no matter what transition sequence is taken.

Verification is achieved by developing proof scores in CafeOBJ . Sufficient verification conditions are formalized for verifying invariants and (`p leads-to q`) properties. For each verification condition, a proof score is constructed to (1) generate a finite set of state patterns that covers all possible infinite states and (2) check validity of the verification condition for all the covering state patterns by reductions.

The method achieves significant automation of proof score developments.

1 Introduction

Constructing specifications and verifying them in the upstream of software development are still one of 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 [9, 10]. In the proof score approach, an executable algebraic specification language (i.e. CafeOBJ [3] in our case) is used to specify a system and system properties, and the reduction (or rewriting) engine of the language is used as a proof engine to prove that the system satisfy the properties of interest.

Proof plans for verifying the system properties are coded into proof scores, and are also written in the algebraic specification language. Usually, a proof score describes modules and predicates in the modules that constitute a sufficient condition for verifying a system property. The language processor interprets the specification and proof score and verifies the validity of the sufficient condition by checking all the predicates with reductions. Logical soundness of the reductions is guaranteed by the fact that the reductions are consistent with the equational reasoning with the equations in the specification and the proof score [11].

The concept of proof supported by proof scores is similar to that of LP [14]. Proof scripts written in tactic languages provided by theorem provers (proof assistants) such as Coq [6] and Isabelle/HOL [19] 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, executable and efficient logical foundation based on the equational and rewriting logics [11, 17].

Effective coordination of inference (*à* la theorem proving, e.g. [6, 15, 19, 23]) and search (*à* la model checking, e.g. [5, 13]) is important for making proof scores more effective and powerful, and we have developed several techniques [22, 10]. The generate & check method described in this paper is a recent development of this kind. The method is based on (1) a state representation as a set of observers, and (2) systematic generation of finite state patterns that cover all possible infinite cases.

The rest of the paper is organized as follows. Section 2 explains necessary mathematical concepts and notations. Section 3 presents system and property specifications of the QLOCK protocol in the CafeOBJ language. Section 4 describes the generate & check method with necessary theoretical expositions. Section 5 presents proof scores for QLOCK through the generate&check method. Section 6 explains related work and future issue.

2 Preliminaries

2.1 Equational Specifications and Quotient Term Algebras

Let $\Sigma = (S, \leq, F)$ be a regular order-sorted signature. Where S is a set of sorts, $\leq$ is a partial order on S, and $F \stackrel{\text{def}}{=} \{F_{s_1 \cdots s_m s}\}_{s_1 \cdots s_m s \in S^+}$ is S^+-sorted set of function symbols. Let $X = \{X_s\}_{s \in S}$ be an S-sorted set of variables, then the **S-sorted set of $\Sigma(X)$-term** is defined inductively as follows. Regularity of an order-sorted signature guarantees the existence of the least sort of a term and makes the definition consistent [12].

- each constant $f \in F_s$ is a $\Sigma(X)$-term of sort s,
- each variable $x \in X_s$ is a $\Sigma(X)$-term of sort s,
- t is a $\Sigma(X)$-term of sort s' if t is a $\Sigma(X)$-term of sort s and $s < s'$, and
- $f(t_1, \ldots, t_n)$ is a $\Sigma(X)$-term of sort s for each operator $f \in F_{s_1 \ldots s_n s}$ and $\Sigma(X)$-terms t_i of sort s_i for $i \in \{1, \ldots, n\}$.

Let $T_\Sigma(X)_s$ denote a set of $\Sigma(X)$-terms of sort s, and let $T_\Sigma(X) \stackrel{\text{def}}{=} \{T_\Sigma(X)_s\}_{s \in S}$, and let $T_\Sigma \stackrel{\text{def}}{=} T_\Sigma(\{\})$. $T_\Sigma(X)$ is called an S-sorted set of $\Sigma(X)$-terms, and T_Σ is called an S-sorted set of Σ-terms. A Σ-term is also called a ground term or a term without variables. Both of $T_\Sigma(X)$ and T_Σ can be organized as Σ-algebras in the obvious way by using the above inductive definition of $\Sigma(X)$-terms. For an S-sorted set T, let $(t \in T) \stackrel{\text{def}}{=} (\exists s \in S)(s \in T_s)$.

Let $l, r \in T_\Sigma(X)_s$ for some $s \in S$ and $c \in T_\Sigma(X)_{\texttt{Bool}}$ for `Bool` $\stackrel{\text{def}}{=} \{\texttt{true}, \texttt{false}\}$, a (conditional) Σ-equation is defined as a sentence of the form $(\forall X)$ $(l = r \ \texttt{if} \ c)$. If the condition c is `true`, the equation is called unconditional and written as $(\forall X)$ $(l = r)$.

For a finite set of equations $E = \{e_1, \cdots, e_n\}$, (Σ, E) represents an equational specification. (Σ, E) defines an order-sorted quotient term algebra $T_\Sigma/\!=_E \stackrel{\text{def}}{=} \{(T_\Sigma)_s/(=_E)_s\}_{s \in S}$, where E defines an order-sorted congruence relation $=_E \stackrel{\text{def}}{=} \{(=_E)_s\}_{s \in S}$ on $T_\Sigma = \{T_{\Sigma s}\}_{s \in S}$. Note that if $e_i = (\forall X)(l_i = r_i \ \texttt{if} \ c_i)$ for $i \in \{1, \cdots, n\}$ and Y is disjoint from X, then $T_\Sigma(Y)/\!=_E$ can be defined similarly by interpreting $T_\Sigma(Y)$ as $T_{\Sigma \cup Y}$. Where $\Sigma \cup Y$ is a signature obtained by interpreting Y as an order-sorted set of fresh constants.

Proof scores in CafeOBJ are mainly developed for equational specification (Σ, E) (i.e. for $T_\Sigma/\!=_E$). Note that the description of $T_\Sigma/\!=_E$ here is quite casual, and refer to [11] for more detailed and precise descriptions including constructor-based signatures, models and satisfaction, equational and specification calculi.

2.2 Rewrite Rules and Reductions

If each variable in r or c is a variable in l, in symbols $(\forall Y)$ $(l \in T_\Sigma(Y)$ `implies` $r, c \in T_\Sigma(Y))$, and l is not a variable, an equation $(\forall X)(l = r \ \texttt{if} \ c)$ can be interpreted as a rewrite rule $(\forall X)(l \rightarrow r \ \texttt{if} \ c)$. Given a set of Σ-equations E that can be interpreted as a set of rewrite rules, the equational specification (Σ, E) defines the one step rewrite relation $\rightarrow_E$ on T_Σ. Note that the definition of $\rightarrow_E$ is not trivial because some rule in E may have a condition (see Section 2.2 of [18] or [26] for details).

The reduction (or rewriting) defined by (Σ, E) is the transitive and reflective closure $\rightarrow_E^*$ of $\rightarrow_E$. In CafeOBJ each equation is interpreted as a rewrite rule, and the reduction is used to check validity of predicates. The following is a fundamental lemma about $=_E$ and $\rightarrow_E^*$.

Lemma 1 [Reduction Lemma] $(\forall t, t' \in T_\Sigma)((t \rightarrow_E^* t') \ \texttt{implies} \ (t =_E t'))$ $\Box$

Let $\theta \in T_\Sigma(Y)^X$ be a substitution (i.e. a map) from X to $T_\Sigma(Y)$ for disjoint X and Y then θ extends to a morphism from $T_\Sigma(X)$ to $T_\Sigma(Y)$, and $t\,\theta$ is the term obtained by substituting $x \in X$ in t with $x\,\theta$.

The following lemma about the reduction plays an important role in the generate & check method.

Lemma 2 [Substitution Lemma]

$$(\forall p \in T_\Sigma(X)_{\texttt{Bool}})((p \rightarrow_E^* \texttt{true}) \ \texttt{implies} \ (\forall \theta \in T_\Sigma(Y)^X)(p\,\theta \rightarrow_E^* \texttt{true}))$$

and

$$(\forall p \in T_\Sigma(X)_{\texttt{Bool}})((p \rightarrow_E^* \texttt{false}) \ \texttt{implies} \ (\forall \theta \in T_\Sigma(Y)^X)(p\,\theta \rightarrow_E^* \texttt{false}))$$

where each $x \in X$ in p and each $y \in Y$ in $p\,\theta$ are treated as fresh constants in the reductions $(p \rightarrow_E^* \texttt{true})$, $(p \rightarrow_E^* \texttt{false})$ and $(p\,\theta \rightarrow_E^* \texttt{true})$, $(p\,\theta \rightarrow_E^* \texttt{false})$ respectively.

Proof Sketch: $((p \rightarrow_E \texttt{true})$ `implies` $(p\,\theta \rightarrow_E \texttt{true}))$ can be proved from the definition of $\rightarrow_E$, and $((p \rightarrow_E^* \texttt{true})$ `implies` $(p\,\theta \rightarrow_E^* \texttt{true}))$ is proved by the induction on the length of the reduction. The `false` case is proved similarly. □

Lemma 1 and Lemma 2 with $Y = \{\}$ imply the following lemma. Where $(\forall X)(p =_E \texttt{true}) \overset{\text{def}}{=} (\forall \theta \in {T_\Sigma}^X)(p\,\theta =_E \texttt{true})$.

Lemma 3 [Lemma of Constants]

$$(\forall p \in T_\Sigma(X)_{\texttt{Bool}})((p \rightarrow_E^* \texttt{true}) \ \texttt{implies} \ (\forall X)(p =_E \texttt{true}))$$

Where each $x \in X$ in p is treated as a fresh constant in the reduction $(p \rightarrow_E^* \texttt{true})$. □

2.3 Transition Systems

It is widely recognized that the majority of systems/problems in many fields can be modeled as transition systems and their invariants.

A transition system (or state machine) is defined as a three tuple (St, Tr, In). St is a set of states, $Tr \subseteq St \times St$ is a set of transitions on the states, and $In \subseteq St$ is a set of initial states. $(s, s') \in Tr$ denotes a transition from the state s to the state s'. A sequence of states $s_1 s_2 \cdots s_n$ with $(s_i, s_{i+1}) \in Tr$ for $i \in \{1, \cdots, n-1\}$ is called a **transition sequence**[1]. A state $s_r \in St$ is defined to be **reachable** if there exists a transition sequence $s_1 s_2 \cdots s_r$ $(r = 1, 2, \cdots)$ such that $s_1 \in In$. A state predicate p (i.e. a function from St to `Bool`) is defined to be an **invariant** (or an invariant property) if $(p(s_r) = \texttt{true})$ for any reachable state s_r.

Let (Σ, E) be an equational specification with an unique topmost sort (i.e. a sort without sub sorts) `State`, and let $tr = (\forall X)(l \rightarrow r \ \texttt{if} \ c)$ be a rewrite rule with $l, r \in T_\Sigma(X)_{\texttt{State}}$ and $c \in T_\Sigma(X)_{\texttt{Bool}}$, then tr is called a transition rule and defines the one step transition relation $\rightarrow_{tr} \in T_\Sigma(Y)_{\texttt{State}} \times T_\Sigma(Y)_{\texttt{State}}$ for Y being disjoint from X as follows.

$$(s \rightarrow_{tr} s') \overset{\text{def}}{=} (\exists \theta \in T_\Sigma(Y)^X)((s = l\,\theta) \ \texttt{and} \ (s' = r\,\theta) \ \texttt{and} \ (c\,\theta =_E \texttt{true}))$$

Note that the rewriting mechanism is used to define the transition relation but it is different from the rewrite relation that defines the one way (or non-commutative) equality. Note also that $=_E$ is understood to be defined with $((\Sigma \cup Y), E)$ by considering $y \in Y$ as a fresh constant if Y is not empty.

Let $TR = \{tr_1, \cdots, tr_m\}$ be a set of transition rules, let $\rightarrow_{TR} \overset{\text{def}}{=} \bigcup_{i=1}^{m} \rightarrow_{tr_i}$, and let $In \subseteq (T_\Sigma)_{\texttt{State}}/(=_E)_{\texttt{State}}$. Then (Σ, E, TR) defines a transition system

[1] "$(s_i, s_{i+1}) \in Tr$ for $i \in \{1, \cdots, 0\}$" is interpreted as `true` and any $s \in St$ is a transition sequence of length 1.

$((T_\Sigma)_{\texttt{State}}/(=_E)_{\texttt{State}}, \rightarrow_{TR}, In)$.[2] A specification (Σ, E, TR) is called a transition specification.

The idea underlies the transition specification (Σ, E, TR) and the transition system $((T_\Sigma)_{\texttt{State}}/(=_E)_{\texttt{State}}, \rightarrow_{TR}, In)$ is same as the one for the topmost rewrite theory [24, 25, 17]. The generate & check method for (Σ, E, TR) is based on, however, only reductions in proof scores.

2.4 Verification of Invariant Properties

Given a transition system $TS = (St, Tr, In)$, let *init* be a state predicate that specifies the initial states (i.e. $(\forall s \in St)$ $(init(s)$ `iff` $(s \in In)))$, and let $p_1, p_2, \cdots, p_n$ $(n \in \{1, 2, \cdots\})$ be state predicates of TS.

Lemma 4 [Invariant Lemma] The following three conditions are sufficient for p_t to be an invariant. Here, $inv(s) \overset{\text{def}}{=} (p_1(s)$ `and` $p_2(s)$ `and` $\cdots$ `and` $p_n(s))$ for $s \in St$.

(1) $(\forall s \in St)(inv(s)$ `implies` $p_t(s))$
(2) $(\forall s \in St)(init(s)$ `implies` $inv(s))$
(3) $(\forall (s, s') \in Tr)(inv(s)$ `implies` $inv(s'))$

Proof: $inv(s_r)$ is proved to be `true` for any reachable state s_r by induction on the length m of a transition sequence from an initial state to s_r. The case of $m = 1$ is deduced by (2). The induction step (i.e. the case $m = k$ implies the case $m = k + 1$) is deduced by (3). Hence *inv* is an invariant, and p_t is deduced to be an invariant by (1) . □

A predicate that satisfies the conditions (2) and (3) like *inv* is called an **inductive invariant**. If p_t itself is an inductive invariant then taking $p_1 = p_t$ and $n = 1$ is enough. However, $p_1, p_2, \cdots, p_n$ $(n > 1)$ are almost always needed to be found for getting an inductive invariant, and to find them is a most difficult part of the invariant verification.

It is worthwhile to note that there are following two contrasting approaches for formalizing $p_1, p_2, \cdots, p_n$ for a transition system and its property p_t.

- Make $p_1, p_2, \cdots, p_n$ as minimal as possible to imply the target property p_t;
  - usually done by lemma finding in interactive theorem proving,
  - it is difficult to find lemmas without some comprehensive understanding of the system.
- Make $p_1, p_2, \cdots, p_n$ as comprehensive as possible to characterize the system;
  - usually done by specifying elemental properties of the system as much as possible in formal specification development,
  - it is difficult to identify the elemental properties without focusing on the property to be proved (i.e. p_t).

[2] Note that $(T_\Sigma)_{\texttt{State}}/(=_E)_{\texttt{State}}$ is better to be understood as $T_\Sigma/{=_E}$, for usually the sort `State` can only be understood together with other related sorts like `Bool`, `Nat`, `Queue`, etc.

3 Specification of QLOCK in CafeOBJ

A simple but non-trivial example is used to explain the generate & check method throughout this paper. It is difficult to include the full specification, and interested readers are encouraged to visit the web page:

`http://www.jaist.ac.jp/~kokichi/misc/1407gcmvtsco/`

The example used is a mutual exclusion protocol QLOCK. A mutual exclusion protocol 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".

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.
  - If the agent finishes to use the common equipment, remove its name from the top of the queue.
– The protocol starts from the state with the empty queue.

3.1 System Specification

OTS (Obervational Transition System) is a scheme for formalizing transition systems. A state is formalized as a collection of typed observed values given by **observers** (or observation operations). A state transition is formalized as an **action** that defines changes of the observed values between the current state and the next state.

For the generate & check method, generations of finite state patterns (i.e. state terms composed of constructors and variables) that subsume all the possible infinite states is a key procedure, and states are assumed to be represented with an appropriate data structure (or configuration). This is different from the original OTS scheme where there is no assumption on the structure of a state [20, 21].

`AOB, AID-QUEUE, and STATE:` A state of QLOCK is defined as a pair of a queue and a set of observers by the following module `STATE`.

```
-- agent observer
mod! AOB {pr(LABEL) pr(AID)
[Aob] op (lb[_]:_) : Aid Label -> Aob {constr} .}
-- queue of Aid
mod! AID-QUEUE {pr(QUEUE(AID{sort Elt -> Aid}))}
-- a state is defined as a pair of a queue of Aid and a set of Aob
mod! STATE{pr(AID-QUEUE) pr(SET(AOB{sort Elt -> Aob})*{sort Set -> Aobs})
-- a state is a pair of Qu and Aobs
[State] op _$_ : Qu Aobs -> State {constr} .}
```

`pr(_)` indicates a **protecting** importation, and declares to import a module without changing its models. The module `LABEL` defines the three labels `rs` (remainder section), `ws` (waiting section), `cs` (critical section) for indicating the status of each agent. The module `AID` defines unbounded number of agent names. The parameterized modules `QUEUE` and `SET` define the data structures needed to define `STATE`. `QUEUE(AID{sort Elt -> Aid})` defines the module obtained by instantiating the parameter `X` of `QUEUE` by `AID` with the interpretation of `Elt` as `Aid`. Similarly `SET(AOB{sort Elt -> Aob})` defines sets of observers. `*{sort Set -> State}` defines the renaming of `Set` to `State`. As a result, a state is presented as a pair of a `Q:Qu` and a set of `(lb[A:Aid]: L:Label)` for all `A:Aid`. Where the term `(lb[A:Aid]: L:Label)` denotes that an agent `A` is in the status `L`.

Let `STATE`-n denote `STATE` with a n agent ids (i.e. `Aid` $= \{\mathsf{a}_1, \cdots, \mathsf{a}_n\}$), and let $\Sigma_{\mathtt{STATE}-n}$ be a signature of `STATE`-n, then the state space of `STATE`-n is defined as $St_{\mathtt{STATE}-n} \overset{\text{def}}{=} T_{\Sigma_{\mathtt{STATE}-n}}$.

`WT`, `TY`, `EX`, and `QLOCKsys`: The QLOCK protocol is defined by the following four modules. The transition rule of the module `TY` indicates that if the top element of the queue is `A:Aid` (i.e. `Qu` is `(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` is `(A & Q)`). The other two transition rules can be read similarly. Note that the module `WT`, `TY`, `EX` formulate the three actions explained in the beginning of the Section 3 precisely and succinctly. `QLOCKsys` is just combining the three modules.

```
-- wt: want transition
mod! WT {pr(STATE)
trans[wt]:    (Q:Qu    $ ((lb[A:Aid]: rs) AS:Aobs))
          => ((Q & A)  $ ((lb[A]:     ws) AS)) . }
-- ty: try transition
mod! TY {pr(STATE)
trans[ty]:   ((A:Aid & Q:Qu) $ ((lb[A]: ws) AS:Aobs))
          => ((A     & Q)    $ ((lb[A]: cs) AS)) . }
-- ex: exit transition
-- this transition can be defined by 'trans' rule with two
-- (A:Aid)s in the left hand side like [ty], 'ctrans' is used
-- here to show an example of conditional transition rule
mod! EX {pr(STATE)
ctrans[ex]:   ((A1:Aid & Q:Qu) $ ((lb[A2:Aid]: cs) AS:Aobs))
          =>           (Q     $ ((lb[A2]:     rs) AS))
             if (A1 = A2) . }
-- system specification of QLOCK
mod! QLOCKsys{pr(WT + TY + EX)}
```

A declaration of a transition rule starts with `trans`, contains rule's name `[_]:`, current state term and next state term are placed before and after `=>` respectively, and ends with "`.`". Note that because a state configuration is a set (i.e. a term composed of associative, commutative, and idempotent binary constructors `(_ _)`) the 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`.

For `STATE-`n with `Aid` $= \{\mathsf{a}_1, \cdots, \mathsf{a}_n\}$, the `trans` or `ctrans` rules `wt,ty,ex` defines the one step transition relations $\rightarrow_{\mathtt{wt}}, \rightarrow_{\mathtt{ty}}, \rightarrow_{\mathtt{ex}}$ respectively on the state space $St_{\mathtt{STATE}-n} = T_{\Sigma_{\mathtt{STATE}-n}}$. `QLOCKsys-`$n$ with `STATE-`n defines a set of transitions $Tr_{\mathtt{QLOCKsys}-n} \subseteq (St_{\mathtt{STATE}-n} \times St_{\mathtt{STATE}-n})$ as $Tr_{\mathtt{QLOCKsys}-n} \overset{\mathrm{def}}{=} (\rightarrow_{\mathtt{wt}} \cup \rightarrow_{\mathtt{ty}} \cup \rightarrow_{\mathtt{ex}})$.

3.2 Property Specification

Property specification is supposed to define the initial state predicate *init* and the possible inductive invariant *inv* in the lemma 4. Both of *init* and *inv* are going to be defined as conjunctions of elemental predicates. For defining the elemental predicates of QLOCK, the module `PNAT` for Peano style unary natural numbers `Nat` with addition `_+_`and greater than `_>_`operations is prepared. Using `PNAT`, fundamental functions on `State` like "the number of a label in a state", "the number of an aid in a state", "the number of an aid in a queue", "label of an agent in a State" are defined. All of these functions are naturally defined with recursive equations.

`INIT`**:** Using the fundamental functions on `State`, elemental state predicates like "at least one agent in a state" (`aoa`), "no duplication of an Aid in a state" (`1a`), "the queue is empty" (`qe`), "any Aid is in `rs` status" (`allRs`) are defined. Using the elemental state predicates, the initial state predicate `init` of QLOCK is defined as follows.

```
-- an initial state predicate
mod! INIT {pr(STATEpred-init)
op init : -> PnameSeq .  eq init = aoa 1a qe allRs .
-- initial state predicate
pred init : State . eq init(S:State) = cj(init,S) . }
```

Note that `cj` is defined as:

```
   op cj : PnameSeq State -> Bool .
   eq cj(PN:Pname PNS:PnameSeq,S:State) = cj(PN,S) and cj(PNS,S) .
```

and a conjunction of predicates is defined as a sequence of `Pname` (i.e. `PnameSeq`).

`INV` and `QLOCKprop`: The target predicate of QLOCK is a mutual exclusion predicate defined as follows.

```
-- mutual exclusion predicate; this is the target predicate
op mx : -> Pname .
eq[mx]: cj(mx,S:State) = (#ls(S,cs) = 0) or (#ls(S,cs) = (s 0)) .
pred mx : State .  eq mx(S:State) = cj(mx,S) .
```

Where, `(#ls(S,cs) = 0) or (#ls(S,cs) = (s 0))` means there is zero or one agent with `cs` status in a state. Elemental state predicates for the possible inductive invariant are selected to specify the statuses like “if queue is empty” (`qep`), “if agent is in rs” (`rs`), “if agent is in ws” (`ws`), “if agent is in cs” (`cs`), “if cs then it should be the top of the queue” (`cst`), and the possible inductive invariant `inv` of QLOCK is defined by the module `INV` as follows. The module `QLOCKprop` for QLOCK property specification is just combining `INIT` and `INV`.

```
-- a possible inductive invariant predicate
mod! INV {pr(STATEpred-inv)
op inv : -> PnameSeq . eq inv = aoa 1a mx qep rs ws cs cst .
pred inv : State .  eq inv(S:State) = cj(inv,S) .
}
-- property specification of QLOCK
mod! QLOCKprop{pr(INIT + INV)}
```

4 Generate & Check Method

The idea underlies the Generate & Check Method is simple and general. Let Srt be a sort and p be a predicate on Srt, then by the lemma 2 (Substitution Lemma)

$$(p(X\!:\!Srt) \rightarrow_E^* \texttt{true}) \texttt{ implies } (\forall t \in (T_\Sigma)_{Srt})(p(t) =_E \texttt{true})$$

holds, and $(p(X\!:\!Srt) \rightarrow_E^* \texttt{true})$ is sufficient to prove $(\forall t)p(t)$. However, usually p is not simple enough to obtain $(p(X : Srt) \rightarrow_E^* \texttt{true})$ directly, and we need to analyze the structure of terms in $(T_\Sigma)_{Srt}$ and E for (1) **generating** a set of terms $\{t_1, \cdots, t_m\} \subseteq T_\Sigma(Y)_{Srt}$ that covers all possible cases of $(T_\Sigma)_{Srt}$, and (2) **checking** $(p(t_i) \rightarrow_E^* \texttt{true})$ for each $i \in \{1, \cdots, m\}$.

Note that **induction** is an already established technique for proving $(p(X : Srt) \rightarrow_E^* \texttt{true})$ for a constrained sort Srt with proof scores [11], and the Generate & Check is another independent technique for coping with sometimes a large number of cases.

4.1 Generate & Check for $\forall st \in St$

A term $t' \in T_\Sigma(Y)$ is defined to be an **instance** of a term $t \in T_\Sigma(X)$ iff there exits a substitution $\theta \in T_\Sigma(Y)^X$ such that $t' = t\,\theta$.

A finite set of terms $C \subseteq T_\Sigma(X)$ is defined to **subsume** a (may be infinite) set of ground terms $G \subseteq T_\Sigma$ iff for any $t' \in G$ there exits $t \in C$ such that t' is an instance of t.

Lemma 5 [Subsume Lemma]
Let a finite set of state terms $C \subseteq T_{\Sigma}(X)_{\texttt{State}}$ subsume the set of all ground state terms $(T_{\Sigma})_{\texttt{State}}$, and let p be a state predicate.

$$((\forall s \in C)(p(s) \to_E^* \texttt{true}))\ \texttt{implies}\ ((\forall t \in (T_{\Sigma})_{\texttt{State}})(p(t) \to_E^* \texttt{true}))$$

Proof: Let $C = \{s_1, \cdots, s_m\}$. Note that $p(s_i) \in T_{\Sigma}(X)_{\texttt{Bool}}$ for any $s_i \in \{s_1, \cdots, s_m\}$. Then, by the definition of "subsume", for any ground state term $t \in (T_{\Sigma})_{\texttt{State}}$, there exits $s_j \in \{s_1, \cdots, s_m\}$ and a substitution $\theta \in T_{\Sigma}^{X}$ such that $t = s_j\theta$. Hence, if $(p(s_i) \to_E^* \texttt{true})$ for all $s_i \in \{s_1, \cdots, s_m\}$ then $(p(s_j) \to_E^* \texttt{true})$, and, by the lemma 2 (Substitution Lemma), $((p(s_j)\theta = p(s_j\theta) = t) \to_E^* \texttt{true})$ holds. □

The lemma 5 and the lemma 1 imply the validity of the following generate & check. Note that $(t_1 \twoheadrightarrow_E^* t_2)$ means that the term t_1 is reduced to the term t_2 by the CafeOBJ's reduction engine, and $(t_1 \twoheadrightarrow_E^* t_2)$ implies $(t_1 \to_E^* t_2)$ but not necessary $(t_1 \to_E^* t_2)$ implies $(t_1 \twoheadrightarrow_E^* t_2)$.

Generate&Check-S Let $((T_{\Sigma})_{\texttt{State}}/(=_E)_{\texttt{State}}, \to_{TR}, In)$ be a transition system defined by a transition specification (Σ, E, TR) (see Section 2.3). The following generate & check are sufficient for verifying

$$(\forall t \in (T_{\Sigma})_{\texttt{State}})(p_{st}(t) =_E \texttt{true})$$

for a state predicate p_{st}.

Generate a finite set of state terms $C \subseteq T_{\Sigma}(X)_{\texttt{State}}$ that subsumes $(T_{\Sigma})_{\texttt{State}}$.
Check $(p_{st}(s) \twoheadrightarrow_E^* \texttt{true})$ for each $s \in C$.

□

4.2 Built-in Search Predicate of CafeOBJ

The verification condition (3) for invariant verification in Lemma 4 contains an universal quantification over the set of transitions Tr, and it is generally difficult to specify Tr as a sort. CafeOBJ's built-in search predicate makes it possible to translate an universal quantification over Tr into an universal quantification over St.

The built-in search predicate is declared as follows.

```
pred _=(*,1)=>+_if_suchThat_{_} : State State Bool Bool Info .
```

`Info` is a sort for showing necessary information. The first argument is the current state `S:State`; the second argument is the variable for binding the found next state `SS:State`; the third argument is the variable for binding the found condition `CC:Bool`; the fourth argument is a predicate `p(S,SS,CC)` whose validity is to be checked; the fifth argument is a term `i(S,SS,CC)` for showing the necessary information. Note that in the use of this predicate the second and third arguments is always variables like `SS:State` and `CC:Bool` for binding the found next state and condition respectively.

Let $((T_\Sigma)_{\texttt{State}}/(=_E)_{\texttt{State}}, \rightarrow_{TR}, In)$ be a transition system defined by a transition specification (Σ, E, TR) (see Section 2.3), and let $TR = \{tr_1, \cdots, tr_m\}$. For a state term $s \in T_\Sigma(Y)_{\texttt{State}}$, the reduction of a Boolean term:

```
s =(*,1)=>+ SS:State if CC:Bool suchThat p(s,SS,CC) {i(s,SS,CC)}
```

with $\twoheadrightarrow_E^* \cup \rightarrow_{TR}$ is defined to behave as follows.

1. Search for every pair (tr_j, θ) of a transition rule $tr_j = (\forall X)(l_j \rightarrow r_j \texttt{ if } c_j)$ in TR and a substitution $\theta \in T_\Sigma(Y)^X$ such that $s = l_j\,\theta$.
2. For each found (tr_j, θ), let ($\texttt{SS} = r_j\,\theta$) and ($\texttt{CC} = c_j\,\theta$) and print out $\texttt{i(}l_j\,\theta$, $r_j\,\theta$, $c_j\,\theta\texttt{)}$ and tr_j if ($\texttt{p(}l_j\,\theta$, $r_j\,\theta$, $c_j\,\theta\texttt{)} \twoheadrightarrow_E^* \texttt{true}$) holds.
3. Returns `true` if some print out exits, and returns `false` otherwise.

4.3 Generate & Check for $\forall tr \in Tr$

Let `q` be a predicate "`pred q : State State`" for stating some relation of the current state and the next state, like ($inv(s)$ `implies` $inv(s')$) in the condition (3) for invariant verification in Lemma 4. Let the predicates `_then_` and `valid-q` be defined as follows in CafeOBJ using the built-in search predicate. Note that `_then_` is different from `_implies_` because (`B:Bool implies true = true`) for `_implies_` but only (`true then true = true`) for `_then_`.

```
pred _then_ : Bool Bool .
eq (true then B:Bool) = B . eq (false then B:Bool) = true .
pred valid-q : State State Bool .
eq valid-q(S:State,SS:State,CC:Bool) =
    not(S =(*,1)=>+ SS if CC suchThat not((CC then q(s,SS)) == true)
        {i(S,SS,CC)}) .
```

For a state term $s \in T_\Sigma(Y)_{\texttt{State}}$, the reduction of the Boolean term:

```
valid-q(s,SS:State,CC:Bool)
```

with $\twoheadrightarrow_E^* \cup \rightarrow_{TR}$ behaves as follows based on the definition of the behavior of the built-in search predicate.

1. Search for evey pair (tr_j, θ) of a transition rule $tr_j = (\forall X)(l_j \rightarrow r_j \texttt{ if } c_j)$ in TR and a substitution $\theta \in T_\Sigma(Y)^X$ such that $s = l_j\,\theta$.
2. For each found (tr_j, θ), let ($\texttt{SS} = r_j\,\theta$) and ($\texttt{CC} = c_j\,\theta$) and print out $\texttt{i(}l_j\,\theta$, $r_j\,\theta$, $c_j\,\theta\texttt{)}$ and tr_j if ($\texttt{not((}c_j\,\theta \texttt{ then q(}l_j\,\theta, r_j\,\theta\texttt{)) == true)} \twoheadrightarrow_E^* \texttt{true}$).
3. Returns `false` if any print out exits, and returns `true` otherwise.

Note that ($\texttt{not((}c_j\,\theta \texttt{ then q(}l_j\,\theta, r_j\,\theta\texttt{)) == true)} \twoheadrightarrow_E^* \texttt{true}$) means ($\texttt{(}c_j\,\theta \texttt{ then q(}l_j\,\theta, r_j\,\theta\texttt{))} \twoheadrightarrow_E^* \texttt{false}$) or ($\texttt{(}c_j\,\theta \texttt{ then q(}l_j\,\theta, r_j\,\theta\texttt{))} \twoheadrightarrow_E^*$ `<not-true-or-false>`), and no print out for (tr_j, θ) means that ($\texttt{(}c_j\,\theta \texttt{ then q(}l_j\,\theta, r_j\,\theta\texttt{))} \twoheadrightarrow_E^* \texttt{true}$). It in turn means ($c_j\,\theta \twoheadrightarrow_E^* \texttt{false}$) or (($c_j\,\theta \twoheadrightarrow_E^* \texttt{true}$) and ($\texttt{q(}l_j\,\theta, r_j\,\theta\texttt{)} \twoheadrightarrow_E^* \texttt{true}$)). Hence, if ($\texttt{valid-q(s,SS:State,CC:Bool)} \twoheadrightarrow_E^* \cup \rightarrow_{TR} \texttt{true}$) for $\texttt{s} \in T_\Sigma(Y)$, (($c_j\,\theta \twoheadrightarrow_E^* \texttt{false}$) or (($c_j\,\theta \twoheadrightarrow_E^* \texttt{true}$) and ($\texttt{q(}l_j\,\theta, r_j\,\theta\texttt{)} \twoheadrightarrow_E^* \texttt{true}$))) for any (tr_j, θ) such that ($s = l_j\,\theta$).

We need the following definition of **cover set** for "Generate & Check for $\forall tr \in Tr$".

Definition 6 [Cover] Let $C \subseteq T_\Sigma(Y)$ and $C' \subseteq T_\Sigma(X)$ be finite sets. C is defined to **cover** C' iff for any ground instance $t'_g \in T_\Sigma$ of any $t' \in C'$, there exits $t \in C$ such that t'_g is an instance of t and t is an instance of t'. $\Box$

The following lemma holds for cover sets.

Lemma 7 [Cover Lemma 1] Let $C' \subseteq T_\Sigma(X)_{\texttt{State}}$ be the set of all the left hand sides of the transition rules in TR, and let $C \subseteq T_\Sigma(Y)$ cover C', then the following holds.

$$
\begin{array}{l}
(\forall \texttt{t} \in C)(\texttt{valid-q(t,SS:State,CC:Bool)} \twoheadrightarrow^*_E \cup \rightarrow_{TR} \texttt{true}) \\
\texttt{implies} \\
(\forall (s, s') \in ((T_\Sigma \times T_\Sigma) \cap \rightarrow_{TR}))(\texttt{q}(s, s') \rightarrow^*_E \texttt{true}))
\end{array}
$$

Proof: For any $(s, s') \in T_\Sigma \times T_\Sigma$, if $(s, s') \in \rightarrow_{TR}$, there exits a transition rule $tr_i = (\forall X)(l_i \rightarrow r_i \;\texttt{if}\; c_i) \in TR$ and a substitution $\theta_s \in T_\Sigma^X$ such that $(s = l_i\,\theta_s)$ and $(s' = r_i\,\theta_s)$ and $(c_i\,\theta_s =_E \texttt{true})$ by the definition of $\rightarrow_{TR}$ (see Section 2.3). Because s is a ground instance of $l_i \in C'$ and C covers C', there exits $t \in C$ such that $(t = l_i\,\theta_t)$ for a substitution $\theta_t \in T_\Sigma(Y)^X$ (i.e. t is an instance of l_i) and $(s = t\,\eta_s)$ for a substitution $\eta_s \in T_\Sigma^Y$ (i.e. s is an instance of t). If we assume $(\texttt{valid-q(t,SS:State,CC:Bool)} \twoheadrightarrow^*_E \cup \rightarrow_{TR} \texttt{true})$, because $(t = l_i\,\theta_t)$ for the substitution $\theta_t \in T_\Sigma(Y)^X$, we get $((c_i\,\theta_t \twoheadrightarrow^*_E \texttt{false})$ or $((c_i\,\theta_t \twoheadrightarrow^*_E \texttt{true})$ and $(\texttt{q(}l_i\,\theta_t,\, r_i\,\theta_t\texttt{)} \twoheadrightarrow^*_E \texttt{true})))$. By using the lemma 2 (Substitution Lemma) with Y for X and $\{\}$ for Y and the fact $(\twoheadrightarrow^*_E \texttt{ implies } \rightarrow^*_E)$, we get $((c_i\,\theta_t\eta_s \rightarrow^*_E \texttt{false})$ or $((c_i\,\theta_t\eta_s \rightarrow^*_E \texttt{true})$ and $(\texttt{q(}l_i\,\theta_t,\, r_i\,\theta_t\texttt{)}\eta_s = \texttt{q(}l_i\,\theta_t\eta_s,\, r_i\,\theta_t\eta_s\texttt{)} \rightarrow^*_E \texttt{true})))$. Because $(c_i\,\theta_t\eta_s = c_i\,\theta_s)$ and $(c_i\,\theta_s =_E \texttt{true})$, $(c_i\,\theta_t\eta_s \rightarrow^*_E \texttt{false})$ can not hold, and we get $(\texttt{q(}l_i\,\theta_t\eta_s,\, r_i\,\theta_t\eta_s\texttt{)} \rightarrow^*_E \texttt{true})$. Because $(l_i\,\theta_t\eta_s = l_i\,\theta_s)$ and $(r_i\,\theta_t\eta_s = r_i\,\theta_s)$, it implies $(\texttt{q(}s,\, s'\texttt{)} \rightarrow^*_E \texttt{true})$.

$$\begin{tikzcd}
l_i \arrow[rrr] \arrow[dd, "\theta_s"'] \arrow[dr, "\theta_t"] & & & r_i \arrow[dl, "\theta_t"'] \arrow[dd, "\theta_s"] & \texttt{if} & & c_i \arrow[dl, "\theta_t"'] \arrow[dd, "\theta_s"] \\
& t = l_i\,\theta_t \arrow[r] \arrow[dl, "\eta_s"] & r_i\,\theta_t \arrow[dr, "\eta_s"'] & & \texttt{if} & c_i\,\theta_t \arrow[dr, "\eta_s"'] & \\
s = l_i\,\theta_s \arrow[rrr] & & & s' = r_i\,\theta_s & \texttt{if} & & c_i\,\theta_s =_E \texttt{true}
\end{tikzcd}$$

Fig. 1. Commutative Diagrams for the Cover Lemma 1

$\Box$

The lemma 7 and the lemma 1 imply the validity of the following generate & check.

Generate&Check-T1 Let $((T_\Sigma)_{\texttt{State}}/(=_E)_{\texttt{State}}, \rightarrow_{TR}, In)$ be a transition system defined by a transition specification (Σ, E, TR) (see Section 2.3), and let $C' \subseteq T_\Sigma(X)$ be the set of all the left hand sides of the transition rules in TR. The following generate & check are sufficient for verifying

$$(\forall(s,s') \in ((T_\Sigma \times T_\Sigma) \cap \rightarrow_{TR}))(\texttt{q}_{\texttt{tr}}(s,s') =_E \texttt{true})$$

for a predicate "`pred` $\texttt{q}_{\texttt{tr}}$ `: State State`".

Generate a finite set of state terms $C \subseteq T_\Sigma(Y)_{\texttt{State}}$ that covers C'.
Check $(\texttt{valid-q}_{\texttt{tr}}\texttt{(t,SS:State,CC:Bool)} \twoheadrightarrow_E^* \cup \rightarrow_{TR} \texttt{true})$ for each $\texttt{t} \in C$.

□

By investigating the proof of the lemma 7, it is seen that the following lemma holds.

Lemma 8 [Cover Lemma 2] Let $TR = \{tr_1, \cdots, tr_m\}$ be a set of transition rules. For $i \in \{1, \cdots, m\}$, let $tr_i = (\forall X)(l_i \rightarrow r_i \texttt{ if } c_i)$ and let $C_i \subseteq T_\Sigma(Y)$ cover $\{l_i\}$. Then the following holds.

$(\forall i \in \{1, \cdots, m\})(\forall \texttt{t} \in C_i)(\texttt{valid-q(t,SS:State,CC:Bool)} \twoheadrightarrow_E^* \cup \rightarrow_{tr_i} \texttt{true})$
`implies`
$(\forall(s,s') \in ((T_\Sigma \times T_\Sigma) \cap \rightarrow_{TR}))(\texttt{q}(s,s') \rightarrow_E^* \texttt{true})$

□

The lemma 8 and the lemma 1 imply the validity of the following generate & check.

Generate&Check-T2 Let $TR = \{tr_1, \cdots, tr_m\}$ be a set of transition rules, and let $tr_i = (\forall X)(l_i \rightarrow r_i \texttt{ if } c_i)$ for $i \in \{1, \cdots, m\}$. Then doing the following generate & check for all of $i \in \{1, \cdots, m\}$ is sufficient for verifying

$$(\forall(s,s') \in ((T_\Sigma \times T_\Sigma) \cap \rightarrow_{TR}))(\texttt{q}_{\texttt{tr}}(s,s') =_E \texttt{true})$$

for a predicate "`pred` $\texttt{q}_{\texttt{tr}}$ `: State State`".

Generate a finite set of state terms $C_i \subseteq T_\Sigma(Y)_{\texttt{State}}$ that covers $\{l_i\}$.
Check $(\texttt{valid-q}_{\texttt{tr}}\texttt{(t,SS:State,CC:Bool)} \twoheadrightarrow_E^* \cup \rightarrow_{tr_i} \texttt{true})$ for each $\texttt{t} \in C$. □

4.4 Generate & Check for Verification of Invariant Properties

The condition (1) and (2) of Lemma 4 can be verified by using Generate&Check-S with $\texttt{p}_{\texttt{st}}(s)$ defined as follows.

(1) $\texttt{p}_{\texttt{st}}(s) = (inv(s) \texttt{ implies } p_t(s))$
(2) $\texttt{p}_{\texttt{st}}(s) = (init(s) \texttt{ implies } inv(s))$

Note that, if $inv \overset{\text{def}}{=} (p_1 \; \texttt{and} \cdots \texttt{and} \; p_n)$, usually $p_t = (p_{i_1} \; \texttt{and} \cdots \texttt{and} \; p_{i_m})$ for $\{i_1, \cdots, i_m\} \subseteq \{1, \cdots, n\}$, and (1) is directly obtained and no need to use Generate&Check-S.

The condition (3) of Lemma 4 can be verified by using Generate&Check-T1 or T2 with $\texttt{q}_{\texttt{tr}}(s, s')$ defined as follows.

$$(3)\;\; \texttt{q}_{\texttt{tr}}(s, s') = (inv(s) \; \texttt{implies} \; inv(s'))$$

4.5 Verification of (p leads-to q) Properties

Invariants are fundamentally important properties of transition systems. They are asserting that something bad will not happen (i.e. safety property). However, it is sometimes also important to assert that something good will surely happen (i.e. liveness property). A (p leads-to q) property is a liveness property defined as follows.

Definition 9 [p leads-to q] Let $TS = (St, Tr, In)$ be a transition system, let Rst be the set of reachable states of TS, let $Tseq$ be the set of transition sequences of TS, and let p, q be predicates with arity $(St, Data)$ of TS, where $Data$ is a data sort needed to specify p, q[3]. Then (p leads-to q) is defined to be valid for TS iff the following holds. Where St^+ denotes the set of state sequences with length more than zero, and $s \in \alpha$ means that s is an element in α for $\alpha \in St^+$.

$$\begin{array}{l}(\forall s\alpha \in Tseq)(\forall d \in Data) \\ \quad (((s \in Rst) \; \texttt{and} \; p(s, d) \; \texttt{and} \; (\forall s' \in s\alpha)(\texttt{not} \; q(s', d))) \\ \quad\; \texttt{implies} \\ \quad (\exists \beta t \in St^+)(q(t, d) \; \texttt{and} \; s\alpha\beta t \in Tseq))\end{array}$$

It means that the system will get into a state t with $q(t, d)$ from a state s with $p(s, d)$ no matter what transition sequence is taken. □

The (p leads-to q) property is adopted from the UNITY logic [4], the above definition is, however, not the same as the original one. In the UNITY logic, the basic model is the parallel program with parallel assignments, and (p leads-to q) is defined through applications of inference rules.

It is worthwhile to note that $(s \in Rst)$ is assumed and the invariant property underlies the (p leads-to q) property.

Lemma 10 [p leads-to q] Based on the original transition system $TS = (St, Tr, ln)$, let $\widehat{St} \overset{\text{def}}{=} St \times Data$, let $(((s, d), (s', d)) \in \widehat{Tr}) \overset{\text{def}}{=} ((s, s') \in Tr)$, let $\widehat{In} \overset{\text{def}}{=} Ln \times Data$, and let $\widehat{TS} \overset{\text{def}}{=} (\widehat{St}, \widehat{Tr}, \widehat{Ln})$. Let inv be an invariant of $\widehat{TS}$ and let m be a function from $\widehat{St}$ to $\texttt{Nat}$ (a set of natural numbers), then the following 4

[3] We may need some $Date$ for specifying a predicate on a transition system like "agent with the name N is working" (N is $Data$).

conditions are sufficient for the property (p leads-to q) to be valid for $\widehat{TS}$. Here $\widehat{s} \overset{\text{def}}{=} (s, d)$ for any $d \in \mathit{Data}$, $p(\widehat{s}) \overset{\text{def}}{=} p(s, d)$ and $q(\widehat{s}) \overset{\text{def}}{=} q(s, d)$.

(1) $(\forall(\widehat{s}, \widehat{s'}) \in \widehat{Tr})$
$((\mathit{inv}(\widehat{s})$ `and` $p(\widehat{s})$ `and` $($`not` $q(\widehat{s})))$ `implies` $(p(\widehat{s'})$ `or` $q(\widehat{s'})))$

(2) $(\forall(\widehat{s}, s') \in \widehat{Tr})$
$((\mathit{inv}(\widehat{s})$ `and` $p(\widehat{s})$ `and` $($`not` $q(\widehat{s})))$ `implies` $(m(\widehat{s}) > m(\widehat{s'})))$

(3) $(\forall \widehat{s} \in \widehat{St})$
$((\mathit{inv}(\widehat{s})$ `and` $p(\widehat{s})$ `and` $($`not` $q(\widehat{s})))$ `implies` $(\exists \widehat{s'} \in \widehat{St})((\widehat{s}, \widehat{s'}) \in \widehat{Tr}))$

(4) $(\forall \widehat{s} \in \widehat{St})$
$((\mathit{inv}(\widehat{s})$ `and` $(p(\widehat{s})$ `or` $q(\widehat{s}))$ `and` $(m(\widehat{s}) = 0))$ `implies` $q(\widehat{s}))$

Proof: Note that $\widehat{\ }$s are omitted in the following. The condition (1) asserts that if $(p(s)$ `and` $($`not` $q(s)))$ for any reachable state s, and $($`not` $q(s')))$ for any next state s' of s, then $p(s')$. This implies, by induction on the length of a transition sequence from s, that for any transition sequence $s\alpha \in \mathit{Tseq}$, if $(\forall s' \in s\alpha)($`not` $q(s'))$ then $(\forall s' \in s\alpha)(p(s'))$. It means $p(s')$ keeps to hold while $($`not` $q(s'))$. (2) asserts that $m(s)$ decreases properly for any next state s' of s, if $(p(s)$`and`$($`not` $q(s)))$. (3) asserts that a next state exits while $(p(s')$`and`$($`not` $q(s')))$. Hence, (2) and (3) imply that $m(s')$ keeps to decease properly while $($`not` $q(s'))$, but $m(s')$ is a natural number and should stop to decrease in finite steps, and should get to the state t with $((p(t)$ `or` $q(t))$ `and` $(m(t) = 0))$. (4) asserts that $((p(t)$ `or` $q(t))$ `and` $(m(t) = 0))$ implies $q(t)$. Hence, $(\exists \beta t \in St^{+})(q(t)$ `and` $s\alpha\beta t \in \mathit{Tseq})$). □

4.6 Generat & Check for Verification of (p leads-to q) Properties

The condition (1) and (2) of Lemma 10 can be verified by using Generate &Check-T1 or T2 in Section 4.3 with $\mathtt{q_{tr}}(s, s')$ defined as follows[4].

(1) $\mathtt{q_{tr}}(s, s') = ((\mathit{inv}(s)$ `and` $p(s)$ `and` $($`not` $q(s)))$ `implies` $(p(s')$ `or` $q(s')))$
(2) $\mathtt{q_{tr}}(s, s') = ((\mathit{inv}(s)$ `and` $p(s)$ `and` $($`not` $q(s)))$ `implies` $(m(s) > m(s')))$

The condition (3) and (4) of Lemma 10 can be verified by using Generate &Check-S in Section 4.1 with $\mathtt{p_{st}}(s)$ defined as follows.

(3) $\mathtt{p_{st}}(s) = ((\mathit{inv}(s)$ `and` $p(s)$ `and` $($`not` $q(s)))$ `implies` $(s$ `=(*,1)=+ SS:State`$))$
(4) $\mathtt{p_{st}}(s) = ((\mathit{inv}(s)$ `and` $(p(s)$ `or` $q(s))$ `and` $(m(s) = 0))$ `implies` $q(s))$

Note that $(s$ `=(*,1)=+ SS:State`$)$ is a built-in search predicate that returns `true` if there exits $s' \in St$ such that $(s, s') \in Tr$.

5 Proof Scores for QLOCK

Interested readers are encouraged to visit the web page:

`http://www.jaist.ac.jp/~kokichi/misc/1407gcmvtsco/`

for the full proof scores.

[4] $\widehat{\ }$s are omitted.

5.1 Proof Scores for Invariant Properties

The module `INV` of the QLOCK property specification in Section 3.2 defines a possible inductive invariant `inv` as the conjunction of seven state predicates as follows.

```
      eq inv = aoa 1a mx qep rs ws cs cst .
```

The target predicate for QLOCK is `mx`, and the condition (1) of Lemma 4 is proved directly.

Proof Scores for $(\forall s \in St)(init(s)$ `implies` $inv(s))$: The condition (2) of Lemma 4 for QLOCK is verified by using Generate&Check-S of Section 4.1.

Just generating a set $C \subseteq T_\Sigma(Y)_{\texttt{State}}$ that subsumes $(T_\Sigma)_{\texttt{State}}$ or covers a set $C' \subseteq T_\Sigma(X)_{\texttt{State}}$ is trivial. You can take $C = \{S{:}\texttt{State}\}$ for the subsuming and $C = C'[X \to Y]$ for the covering. The challenging part is to guarantee that for the target predicate p the check $(p(t_i) \twoheadrightarrow_E^* \texttt{true})$ for each $t_i \in C$ is successful. Let a set $C \subseteq T_\Sigma(Y)_{\texttt{State}}$ be called p-effective iff the check $(p(t_i) \twoheadrightarrow_E^* \texttt{true})$ for each $t_i \in C$ is successful.

The following set of state patterns {`s1,s2,...,s7`} covers the singleton set of the most general state pattern {`(Q:Qu $ AS:Aobs)`} and subsumes the set of all the ground state terms $(T_\Sigma)_{\texttt{State}}$.

```
    --> case[1]: S:State = (Q:Qu $ empty)
    eq s1  = (q $ empty) .
    --> case[2]: S:State = (Q:Qu $ ((lb[A:Aid]: L:Label) AS:Aobs))
    eq s2  = (empQ $ ((lb[a1]: rs) as)) .      -- wt
    eq s3  = (empQ $ ((lb[a1]: ws) as)) .
    eq s4  = (empQ $ ((lb[a1]: cs) as)) .
    --
    eq s5  = ((a1 & q) $ ((lb[a2]: rs) as)) .  -- wt
    eq s6  = ((a1 & q) $ ((lb[a2]: ws) as)) .  -- ty
    eq s7  = ((a1 & q) $ ((lb[a2]: cs) as)) .  -- ex
}
```

Here, `q` is a variable of sort `Qu`, `as` is a variable of sort `Aobs`, `a1, a2` are variables of sort `Aid`. Note that variables `q, as, a1, a2` are appearing in the terms to be reduced and are declared as fresh constants in the proof scores in CafeOBJ.

It is easy to see that there is no overlap among `s1,s2,...,s7` and they list up all the state patterns with (1) the `Qu` part is `empQ` or `(A1:Aid & Q:Qu)`, and (2) the `Aobs` part is `empty` or `((lb[A:Aid]: L:Label) as:Aobs)`.

Let `init-c` be defined as:

```
      pred init-c : State .
      eq init-c(S:State) = init(S) implies inv(S) .
```

then it is shown that {`s1,s2,...,s7`} is a (`init-c`)-effective set by checking (`init-c(si)` $\twoheadrightarrow_E^*$ `true`) (i.e. "`red init-c(si) .`" returns `true`) for each `si` $\in$ {`s1,s2,...,s7`}.

The cover set {`s1,s2,...,s7`} can be generated by the following combinatorial generation script.

```
[(tg(2)[q,empty])]
       ||
[(tg(2)[(empQ),(tg(1)[(a1),(rs;ws;cs),(as)])])]
       ||
[(tg(2)[(a1 & q),(tg(1)[(a2),(rs;ws;cs),(as)])])]

-- t(1)/tg(1) and t(2)/tg(2) construct state terms
-- defined by the following two equations:
eq t(1)(A:Aid,L:Label,as:Aobs) = ((lb[A]: L) as) .
eq t(2)(Q:Qu,AS:Aobs) = (Q $ AS) .
```

Using the combinatorial generation of the cover set, Generate&Check-S for the state predicate `init-c` can be done automatically by one reduction command in CafeOBJ.

Proof Scores for $(\forall(s,s') \in Tr)(inv(s)$ `implies` $inv(s'))$**:** The condition (3) of Lemma 4 for QLOCK is verified by using Generate&Check-T2 of Section 4.3.

In the QLOCK specification, the three transition rules `wt, ty, ex` are defined in the module `WT, TY, EX` in Section 3.1 and the three left hand sides of the transition rules are as follows.

```
l1 =          (Q:Qu  $ ((lb[A:Aid]:  rs) as:Aobs))
l2 =  ((A:Aid & Q:Qu) $ ((lb[A]:      ws) as:Aobs))
l3 = ((A1:Aid & Q:Qu) $ ((lb[A2:Aid]: cs) as:Aobs))
```

Hence, a minimal set that covers {`l1,l2,l3`} can be obtained as follows.

```
-- State patterns
ops t1 t2 t3 t4 t5 t6 : -> State .
-- covering l1
eq t1 = (empQ $ ((lb[b1]: rs) as)) .      -- wt
eq t2 = ((b1 & q) $ ((lb[b1]: rs) as)) .  -- wt
eq t3 = ((b1 & q) $ ((lb[b2]: rs) as)) .  -- wt
-- covering l2
eq t4 = ((b1 & q) $ ((lb[b1]: ws) as)) .  -- ty
-- covering l2
eq t5 = ((b1 & q) $ ((lb[b1]: cs) as)) .  -- ex
eq t6 = ((b1 & q) $ ((lb[b2]: cs) as)) .  -- ex
```

Here, `b1, b2, b3` are literal variables of sort `Aid`. **Literal variables** are defined to be variables which obey the rule that different literals variables denote different objects, and literal variables `b1, b2, b3` denote different elements of sort `Aid`. Variables `q, as, b1, b2, b3` are also declared as fresh constants. Let `q` and `inv-c` be defined as follows, where `valid-q` is defined as in Section 4.3.

```
pred q : State State .
eq q(S:State,SS:State) = (inv(S) implies inv(SS)) .
--
pred inv-c : State State Bool .
eq inv-c(S:State,SS:State,CC:Bool) = valid-q(S,SS,CC) .
```

Let a set $C \subseteq T_\Sigma(Y)_{\texttt{State}}$ be called **p-effective with TR** iff the check $(p(t_i) \twoheadrightarrow^*_E \cup \rightarrow_{TR}$ `true`) for each $t_i \in C$ is successful. Then it is shown that (1) {`t1, t2, t3`} is (`inv-c`)-effective with {`wt`}, (2) {`t4`} is (`inv-c`)-effective with {`ty`}, and (3) {`t5, t6`} is (`inv-c`)-effective with {`ex`}. This implies that {`t1,...,t6`} is (`inv-c`)-effective with {`wt,ty,ex`}.

The cover set {`t1,...,t6`} can be generated by the following combinatorial generation script.

```
[(tg(2)[(empQ),(tg(1)[(),(b1),rs,(as)])])]
     ||
[(tg(2)[(b1 & q),(tg(1)[(),(b1),(rs;ws;cs),(as)])])]
     ||
[(tg(2)[(b1 & q),(tg(1)[(),(b2),(rs;cs),(as)])])]
```

Using the combinatorial generation of the cover set, Generate&Check-T1 for showing that {`t1,...,t6`} is (`inv-c`)-effective with {`wt,ty,ex`} can be done automatically by one reduction command in CafeOBJ.

5.2 Proof Scores for a (p leads-to q) Property

QLOCK has an interesting (p leads-to q) property. Let `lags` be defined as follows; where `aos` is a destructor for getting the `Aobs` part from a `State`.

```
-- label of an agent in a Aobs
op laga : Aobs Aid -> Label .
eq laga(((lb[A1:Aid]: L:Label) AS:Aobs),A2:Aid) =
   if (A1 = A2) then L else laga((AS),A2) fi .
-- label of an agent in a State
op lags : State Aid -> Label .
eq lags(S:State,A:Aid) = laga(aos(S),A) .
```

Then QLOCK enjoys ((`lags(S,A) = ws`) leads-to (`lags(S,A) = cs`)) property. That is, if an agent gets into the queue, it will get to the top of the queue with "`cs`" label.

If we can identify inv and m of the Lemma 10, proof scores for verifying this property can be developed by using the generate & check method as shown in Section 4.6. It turns out that it is sufficient to take (inv(`S:State`) = `inv(S)`) and (m(`S:State`,`A:Aid`) = `#dms(S,A)`). Where, `inv` is the state predicate proved to be an inductive invariant in Section 5.1, and `#dms` is defined as follows.

```
-- the number of a label in a Aobs
op #lss : Aobs Label -> Nat .
eq #lss(empty,L:Label) = 0 .
eq #lss(((lb[A:Aid]: L1:Label) AS:Aobs),L2:Label) =
   if (L1 = L2) then (s 0) + #lss((AS),L2)
   else #lss((AS),L2) fi .
-- the number of a label in a state
op #ls : State Label -> Nat .
eq #ls(S:State,L:Label) = #lss(aos(S),L) .
-- the depth of the first appearence of an aid in a queue
```

```
op #daq : Qu Aid -> Nat .
eq #daq(A1:Aid & Q:Qu,A2:Aid) =
   if (A1 = A2) then 0 else s(#daq(Q,A2)) fi .
-- counter count of cs
op #ccs : State -> Nat .
eq #ccs(S:State) = if (#ls(S,cs) > 0) then 0 else (s 0) fi .
-- decreasing Nat measure for the lockout freedom verification
op #dms : State Aid -> Nat .
eq #dms(S:State,A:Aid) = ((s s s 0) * #daq(qu(S),A))
                         + #ls(S,rs) + #ccs(S) .
```

The combinatorial generation scripts used are almost same as ones used in Section 5.1, except we need to generate for the pattern (`Q:Qu $ AS:Aobs, A:Aid`) instead of for the pattern (`Q:Qu $ AS:Aobs`).

6 Related Work and Conclusion

Related work: Verification methods for transition systems are largely classified into deductive and algorithmic ones. Majority of the deductive methods are applications of theorem proving methods/systems [6, 15, 19, 23] to verifications of concurrent systems or distributed protocols with infinite states. Most dominant algorithmic methods are based on model checking methods/systems [2, 5] and are targeting to automatic verifications of temporal properties of finite state transition systems. The generate & check method described in this paper is a deductive method with algorithmic combinatorial generations of cover sets. Moreover, reduction is only one deductive mechanism, and it makes theories and proof scores for the method simple and transparent.

Maude [16] is a sister language of CafeOBJ and both languages share many important features. Maude's basic logic is rewriting logic [17] and verification of transition systems with Maude focuses on sophisticated model checking with a powerful associative and/or commutative rewriting engine. There are recent attempts to extend the model checking with Maude for verifying infinite state transition systems [1, 8]. They are based on narrowing with unification, whereas the generate & check method is based on cover sets with ordinary matching and reduction.

Searches on time versus space: There are quite a few researches on search techniques in model checking [5, 13]. It is interesting to observe that what we have done for the generate & check method in this paper is a search in state space with the built-in search predicate that amounts to the complete search across all one step transitions, whereas the search for model checking is along time axis (i.e. transition sequences) as shown in Figure 2.

Future Issue: This paper only describes CafeOBJ specifications and proof scores for the rather small QLOCK example. We have, however, already checked

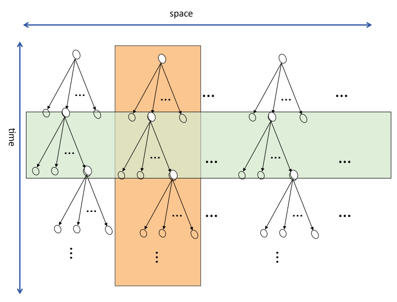

Fig. 2. Searches on Time versus Space

that the method proposed can be applied to more larger examples like ABP (Alternating Bit Protocol [21]). As a matter of fact, the generate & check method should be more important for large problems, for it is difficult to do case analyses manually for them. Once a state configuration is properly designed, large number of patterns (i.e. elements of a cover set) that cover all possible cases 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 developing practical methods to obtain effective cover sets.

Acknowledgments

This work was supported in part by Grant-in-Aid for Scientific Research (S) 23220002 from Japan Society for the Promotion of Science (JSPS).

References

1. Bae, K., Escobar, S., Meseguer, J.: Abstract logical model checking of infinite-state systems using narrowing. In: van Raamsdonk, F. (ed.) RTA. LIPIcs, vol. 21, pp. 81–96. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2013)
2. Baier, C., Katoen, J.P.: Principles of model checking. MIT Press (2008)
3. CafeOBJ: http://www.ldl.jaist.ac.jp/cafeobj/ (2014)
4. Chandy, K.M., Misra, J.: Parallel program design - a foundation. Addison-Wesley (1989)
5. Clarke, E.M., Grumberg, O., Peled, D.: Model checking. MIT Press (2001)
6. Coq: http://coq.inria.fr (2014)
7. 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)
8. Escobar, S., Meseguer, J.: Symbolic model checking of infinite-state systems using narrowing. In: Baader, F. (ed.) RTA. Lecture Notes in Computer Science, vol. 4533, pp. 153–168. Springer (2007)

9. 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)
10. Futatsugi, K.: Fostering proof scores in CafeOBJ. In: Dong and Zhu [7], pp. 1–20
11. Futatsugi, K., Gâinâ, D., Ogata, K.: Principles of proof scores in CafeOBJ. Theor. Comput. Sci. 464, 90–112 (2012)
12. Goguen, J.A., Meseguer, J.: Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Theor. Comput. Sci. 105(2), 217–273 (1992)
13. Grumberg, O., Veith, H. (eds.): 25 Years of Model Checking - History, Achievements, Perspectives, Lecture Notes in Computer Science, vol. 5000. Springer (2008)
14. 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)
15. HOL: http://hol.sourceforge.net (2014)
16. Maude: http://maude.cs.uiuc.edu/ (2014)
17. Meseguer, J.: Twenty years of rewriting logic. J. Log. Algebr. Program. 81(7-8), 721–781 (2012)
18. Nakamura, M., Ogata, K., Futatsugi, K.: Incremental proofs of termination, confluence and sufficient completeness of OBJ specifications. In: Iida, S., Meseguer, J., Ogata, K. (eds.) Specification, Algebra, and Software. Lecture Notes in Computer Science, vol. 8373, pp. 92–109. Springer (2014)
19. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
20. 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)
21. Ogata, K., Futatsugi, K.: Simulation-based verification for invariant properties in the OTS/CafeOBJ method. Electr. Notes Theor. Comput. Sci. 201, 127–154 (2008)
22. Ogata, K., Futatsugi, K.: A combination of forward and backward reachability analysis methods. In: Dong and Zhu [7], pp. 501–517
23. PVS: http://pvs.csl.sri.com (2014)
24. Rocha, C., Meseguer, J.: Proving safety properties of rewrite theories. technical report. Tech. rep., University of Illinois at Urbana-Champaign (2010)
25. Rocha, C., Meseguer, J.: Proving safety properties of rewrite theories. In: Corradini, A., Klin, B., Cîrstea, C. (eds.) CALCO. Lecture Notes in Computer Science, vol. 6859, pp. 314–328. Springer (2011)
26. TeReSe (ed.): Term Rewriting Systems, Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press (2003)