Generic Proof Scores for Generate & Check Method in CafeOBJ*

Kokichi Futatsugi

Japan Advanced Institute of Science and Technology (JAIST)
1-1 Asahidai, Nomi, Ishikawa 923-1292, Japan

Abstract. Generic proof scores for the generate & check method in CafeOBJ are described. The generic proof scores codify the generate & check method as parameterized modules in the CafeOBJ language independently of specific systems to which the method applies. Proof scores for a specific system can be obtained by substituting the parameter modules of the parameterized modules with the specification modules of the specific system.

The effectiveness of the generic proof scores is demonstrated by applying them to a simple but non-trivial example.

1 Introduction

Constructing specifications and verifying them in the upstream of system development are still one of the most important challenges in system development and engineering. It is because many critical defects are caused at the phases of domains, requirements, and designs specifications. Proof scores are intended to meet this challenge [8, 9].

A system and the system's properties are specified in an executable algebraic specification language (CafeOBJ [3] in our case). Proof scores are described also in the same specification language for checking whether the system specifications imply the property specifications. Specifications and proof scores are expressed in equations, and the checks are done only by reduction with the equations. The logical soundness of the checks is guaranteed by the fact that the reduction are consistent with the equational reasoning with the equations in the specification [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 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].

The generate & check method is a theorem proving method for transition systems based on (1) generation of finite state patters that cover all possible

* A shortened version of this paper is submitted to Logic, Rewriting, and Concurrency - Festschrift Symposium in Honor of José Meseguer.

infinite states, and (2) checking the validities of verification conditions for each of the finite state patterns[10]. 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 ($\mathrm{p}(s) = \texttt{true}$) the system will get to a state t with ($\mathrm{q}(t) = \texttt{true}$) no matter what transition sequence is taken.

Modularization via parameterization of proof scores is crucial because (a) it helps to identify reusable proof scores, (b) it helps to give good structures to proof scores, and (c) (a)&(b) make proof scores easy to understand and flexible enough for transparent interactive deduction via rewriting and modifications (i.e. interactive verification).

The rest of the paper is organized as follows. Section 2 includes preliminary materials. Section 3 presents the system specifications of QLOCK (a mutual exclusion protocol). Section 4 presents the generic proof scores for the generate & check method. Section 5 presents the property specifications of QLOCK. Section 6 describes the development of the QLOCK proof scores using the generic proof scores presented in Section 4. Section 7 presents related works and future issues.

2 Preliminaries

Section 2.1 to Section 2.10 give a digest of the paper [10] that is necessary for the following sections.

2.1 Equational Specifications and Quotient Term Algebras

Let $\Sigma = (S, \leq, F)$ be a regular order-sorted signature [12], 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.

- 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. $T_\Sigma(X)$ can be organized as $(\Sigma \cup X)$-algebras in the obvious way by using the above inductive definition of $\Sigma(X)$-terms, where $\Sigma \cup X$ is a signature obtained by interpreting X as an order-sorted set of fresh constants. Similarly, T_Σ can be organized as Σ-algebras.

Let $l, r \in T_\Sigma(X)_s$ for some $s \in S$ and $c \in T_\Sigma(X)_{\texttt{Bool}}$ for a special sort `Bool` with the two constructors `true` and `false`, a Σ-equation is defined as a sentence of the form $(\forall X)(l = r \texttt{ if } c)$. If the condition c is the constant predicate `true`, the equation is called unconditional and written as $(\forall X)(l = r)$. An equation that is not unconditional is called conditional. Throughout this paper an equation may be conditional, and the theory and the method presented are valid even if considering conditional equations.

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 $\texttt{=}_E \stackrel{\text{def}}{=} \{(\texttt{=}_E)_s\}_{s \in S}$ on $T_\Sigma = \{T_{\Sigma s}\}_{s \in S}$. Note that if $e_i = (\forall X)(l_i \texttt{ = } r_i \texttt{ if } c_i)$ for $i \in \{1, \cdots, n\}$ and Y is disjoint from X, then $T_\Sigma(Y)/\texttt{=}_E$ 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 an equational specification (Σ, E) (i.e. for $T_\Sigma/=_E$) [11].

2.2 Rewrite Rules and Reductions

If each variable in r or c is a variable in l (i.e. $(\forall Y)(l \in T_\Sigma(Y) \texttt{ implies } r, c \in T_\Sigma(Y))$) and l is not a variable, an equation $(\forall X)(l = r \texttt{ if } c)$ can be interpreted as a rewrite rule $(\forall X)(l \to r \texttt{ if } c)$. 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 $\to_E$ on T_Σ. Note that the definition of $\to_E$ is not trivial because some rule in E may have a condition (see Section 2.2 of [18] or [25] for details).

The reduction (or rewriting) defined by (Σ, E) is the transitive and reflective closure $\to_E^*$ of $\to_E$. 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 $\to_E^*$.

Lemma 1 (Reduction Lemma) $(\forall t, t' \in T_\Sigma)((t \to_E^* t') \texttt{ implies } (t =_E t'))$ □

Note that the Reduction Lemma holds even if the rewriting relation defined by a specification (Σ, E) is not "terminating", "confluent", and "sufficiently complete". These properties of the rewriting relation are desirable but not necessary for the theory and the method presented in this paper.

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)

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

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. □

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

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** 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 each $i \in \{1, \cdots, n-1\}$ is defined to be a **transition sequence**. Note that any $s \in St$ is defined to be a transition sequence of length 1[1]. A state $s^r \in St$ is defined to be **reachable** if there exists a transition sequence $s_1 s_2 \cdots s_n$ with $s_n = s^r$ for $n \in \{1, 2, \cdots\}$ such that $s_1 \in In$. A state predicate p (i.e. a function from St to $\texttt{Bool}$) 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 a unique topmost sort (i.e. a sort without subsorts) $\texttt{State}$, 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.

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

Note 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 $\rightarrow_{TR} \stackrel{\text{def}}{=} \bigcup_{i=1}^m \rightarrow_{tr_i}$, and let $In \subseteq (T_\Sigma)_{\texttt{State}}/(=_E)_{\texttt{State}}$. In is assumed to be defined via a state predicate $init$ that is defined with E, i.e. $(s \in In)\ \texttt{iff}\ (init(s) =_E \texttt{true})$. Then

[1] For the case in which $n = 1$, $s_1 s_2 \cdots s_n$ is s_1 and $\{1, \cdots, 0\}$ is the empty set, and $((s_i, s_{i+1}) \in Tr$ for each $i \in \{1, \cdots, 0\})$ could be interpreted valid.

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

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, and $inv(s) \stackrel{\text{def}}{=} (p_1(s)$ `and` $p_2(s)$ `and` $\cdots$ `and` $p_n(s))$ for $s \in St$.

Lemma 4 (Invariant Lemma) The following three conditions are sufficient for a state predicate p^t to be an invariant.

(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` $inv(s'))$ □

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.

2.5 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 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 a sufficient condition 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 the generate & check method is general enough for applying not only to the sort `State` but also to any sort *Srt*. As a matter of fact, it can be applied in quite a few occasions in which the necessary cases to be analyzed can be covered by a finite set of term patterns of sort *Srt*. This paper only describes a special but most important application to the sort `State`.

Note also 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 method is another independent technique for coping with sometimes a large number of cases.

[2] $(T_\Sigma)_{\texttt{State}}/(=_E)_{\texttt{State}}$ 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.

2.6 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, then the following holds.

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

Lemma 5 and Lemma 1 imply the validity of following **Generate&Check-S**. 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 $(t_1 \rightarrow^*_E t_2)$ but not necessary $(t_1 \rightarrow^*_E t_2)$ implies $(t_1 \twoheadrightarrow^*_E t_2)$.

[Generate&Check-S] 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). Then, for a state predicate p_{st}, doing the following **Generate** and **Check** are sufficient for verifying $(\forall t \in (T_\Sigma)_{\texttt{State}})(p_{st}(t) =_E \texttt{true})$.

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 $s \in C$. $\square$

2.7 Built-in Search Predicate and Generate & Check for $\forall tr \in Tr$

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

The built-in search predicate is declared as follows.

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

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). 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_`.

```
-- information constructor
[Infom] op (ifm _ _ _ _) : State State Bool Bool -> Infom {constr}
-- for checking conditions of ctrans rules
pred _then _ : Bool Bool .
eq (true  then B:Bool) = B . eq (false then B:Bool) = true .
-- predicate to be checked for a State
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) {(ifm S SS CC q(S,SS))}) .
```

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 (see Section 4.2 of [10]). Note that the $\rightarrow_{TR}$ part is only effective for 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 $\texttt{s} = l_j\,\theta$.
2. For each found (tr_j, θ), let $(\texttt{SS} = r_j\,\theta)$ and $(\texttt{CC} = c_j\,\theta)$ and print out `(ifm s SS CC q(s,SS))` and tr_j if $(\texttt{not((CC then q(s,SS)) == true)} \twoheadrightarrow_E^* \texttt{true})$.
3. Returns `false` if any print out exits, and returns `true` otherwise.

The following definition and lemma are needed 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'. □

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.

$$(\forall \texttt{t} \in C)(\texttt{valid-q(t,SS:State,CC:Bool)} \twoheadrightarrow_E^* \cup \rightarrow_{TR} \texttt{true})$$
$$\texttt{implies}$$
$$(\forall (s, s') \in ((T_\Sigma \times T_\Sigma) \cap \rightarrow_{TR}))(\texttt{q}(s, s') \rightarrow_E^* \texttt{true}))$$ □

Lemma 7 and Lemma 1 imply the validity of following **Generate&Check-T1**.

[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. Then doing the following **Generate** and **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 $\texttt{t} \in C$. □

The following lemma also holds for the cover sets.

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`
$(\forall (s, s') \in ((T_\Sigma \times T_\Sigma) \cap \rightarrow_{TR}))(\texttt{q}(s, s') \rightarrow_E^* \texttt{true})$ □

Lemma 8 and Lemma 1 imply the validity of following **Generate&Check-T2**.

[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** and **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 $\texttt{t} \in C$. □

2.8 Generate&Check for Verification of Invariant Properties

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

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

Note that, if $inv \stackrel{\text{def}}{=} (p_1 \texttt{ and} \cdots \texttt{and } p_n)$, 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 condition (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 $\texttt{q}_\texttt{tr-3}(s, s')$ defined as follows.

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

2.9 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 in the premiss of the definition of (p leads-to q) properties.

Lemma 10 (p leads-to q) Based on the original transition system $TS = (St, Tr, In)$, let $\widehat{St} \stackrel{\text{def}}{=} St \times Data$, let $(((s,d),(s',d)) \in \widehat{Tr}) \stackrel{\text{def}}{=} ((s,s') \in Tr)$, let $\widehat{In} \stackrel{\text{def}}{=} In \times Data$, and let $\widehat{TS} \stackrel{\text{def}}{=} (\widehat{St}, \widehat{Tr}, \widehat{In})$. Let inv be an invariant of $\widehat{TS}$ and let m be a function from $\widehat{St}$ to `Nat` (the set of natural numbers), then the following 4 conditions are sufficient for the property (p leads-to q) to be valid for $\widehat{TS}$. Here $\widehat{s} \stackrel{\text{def}}{=} (s,d)$ for any $d \in Data$, $p(\widehat{s}) \stackrel{\text{def}}{=} p(s,d)$ and $q(\widehat{s}) \stackrel{\text{def}}{=} q(s,d)$.

$$\begin{array}{ll}(1) & (\forall(\widehat{s},\widehat{s'}) \in \widehat{Tr})\\ & \quad((inv(\widehat{s})\ \texttt{and}\ p(\widehat{s})\ \texttt{and}\ (\texttt{not}\ q(\widehat{s})))\ \texttt{implies}\ (p(\widehat{s'})\ \texttt{or}\ q(\widehat{s'})))\\ (2) & (\forall(\widehat{s},s') \in \widehat{Tr})\\ & \quad((inv(\widehat{s})\ \texttt{and}\ p(\widehat{s})\ \texttt{and}\ (\texttt{not}\ q(\widehat{s})))\ \texttt{implies}\ (m(\widehat{s}) > m(\widehat{s'})))\\ (3) & (\forall\widehat{s} \in \widehat{St})\\ & \quad((inv(\widehat{s})\ \texttt{and}\ p(\widehat{s})\ \texttt{and}\ (\texttt{not}\ q(\widehat{s})))\ \texttt{implies}\ (\exists\widehat{s'} \in \widehat{St})((\widehat{s},\widehat{s'}) \in \widehat{Tr}))\\ (4) & (\forall\widehat{s} \in \widehat{St})\\ & \quad((inv(\widehat{s})\ \texttt{and}\ (p(\widehat{s})\ \texttt{or}\ q(\widehat{s}))\ \texttt{and}\ (m(\widehat{s}) = 0))\ \texttt{implies}\ q(\widehat{s}))\quad \square\end{array}$$

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

The conditions (1) and (2) of Lemma 10 can be verified by using Generate &Check-T1 or T2 in Section 2.7 with $\texttt{q}_{\texttt{tr-1}}(\widehat{s},\widehat{s'})$ and $\texttt{q}_{\texttt{tr-2}}(\widehat{s},\widehat{s'})$ defined as follows respectively.

$$\begin{array}{ll}(1) & \texttt{q}_{\texttt{tr-1}}(\widehat{s},\widehat{s'}) = ((inv(\widehat{s})\ \texttt{and}\ p(\widehat{s})\ \texttt{and}\ (\texttt{not}\ q(\widehat{s})))\ \texttt{implies}\ (p(\widehat{s'})\ \texttt{or}\ q(\widehat{s'})))\\ (2) & \texttt{q}_{\texttt{tr-2}}(\widehat{s},\widehat{s'}) = ((inv(\widehat{s})\ \texttt{and}\ p(\widehat{s})\ \texttt{and}\ (\texttt{not}\ q(\widehat{s})))\ \texttt{implies}\ (m(\widehat{s}) > m(\widehat{s'})))\end{array}$$

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

The conditions (3) and (4) of Lemma 10 can be verified by using Generate &Check-S in Section 2.6 with $\texttt{p}_{\texttt{st-3}}(\widehat{s})$ and $\texttt{p}_{\texttt{st-4}}(\widehat{s})$ defined as follows respectively.

(3) $\texttt{p}_{\texttt{st-3}}(\widehat{s}) = ((inv(\widehat{s})\ \texttt{and}\ p(\widehat{s})\ \texttt{and}\ (\texttt{not}\ q(\widehat{s})))\ \texttt{implies}\ (\widehat{s}\ \texttt{=(*,1)=+ SS:State}))$
(4) $\texttt{p}_{\texttt{st-4}}(\widehat{s}) = ((inv(\widehat{s})\ \texttt{and}\ (p(\widehat{s})\ \texttt{or}\ q(\widehat{s}))\ \texttt{and}\ (m(\widehat{s}) = 0))\ \texttt{implies}\ q(\widehat{s}))$

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

2.11 QLOCK: a Mutual Exclusion Protocol

A simple but non-trivial example is used throughout this paper. 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 (concurrent mechanism or algorithm) which can achieve the mutual exclusion is called "mutual exclusion protocol".*

QLOCK is realized by using a unique 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 an agent wants to use the common equipment and its name is not in the queue yet, put its name at the bottom of the queue.
  - If an 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.

2.12 System Specifications, Property Specifications, and Proof Scores

For verifying a system, a model of the system should be formalized and described as system specifications that are formal specifications of the behavior of the system. Based on the system specifications, the system's supposed properties are formalized and described as property specifications. Note that the properties we are considering are either (1) invariant properties or (2) (p leads-to q) properties.

Proof scores are developed to verify that the system's supposed properties are deduced from the system specifications.

Section 4 gives the generic proof scores for the generate & check method. Section 3 and Section 5 give the system and property specifications of QLOCK respectively. Section 6 gives proof scores for invariant properties and a (p leads-to q) property of QLOCK by making use of the generic proof scores and the system and property specifications.

The generic proof scores, the QLOCK specifications, and the QLOCK proof scores are organized as a set of files in the CafeOBJ language system and posted at the following web page.

`http://www.jaist.ac.jp/~kokichi/misc/1502gpsgcmco/`

Interested readers are encouraged to look into the files on the web page, for the files contain quite a few comments including the comments on the CafeOBJ language that are not included in this paper.

3 System Specifications of QLOCK[4]

This section gives the system specifications of QLOCK with explanations on the basics of the CafeOBJ language. It is also intended to be preparations for the main section (Section 4) of this paper where the generate & check method is codified in CafeOBJ language.

3.1 OTS , `LABEL, AID`

A system specification of QLOCK is constructed with the OTS modeling scheme. In OTS (Observational Transition System) a state is formalized as a collection of typed observed values given by **observers** (or observation operations), and 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].

For defining a configuration of the QLOCK state, the following modules `LABEL` and `AID` are necessary.

```
-- three labels for indicating status of each agent
mod! LABEL {
-- label literals and labels
[LabelLt < Label]
-- declaration of constructor constants (operators without arguments)
```

[4] The specifications explained in this section is in the file `qlock-sys.cafe` on the web page.

```
ops rs ws cs : -> LabelLt {constr}
** rs: remainder section, ws: waiting section, cs: critical section
-- an equation to declare that the elements of LabelLt are literals
eq (L1:LabelLt = L2:LabelLt) = (L1 == L2) .
}
-- agent identifiers
mod* AID {[Aid]}
** Aid is declared to be any set of agent identifiers
```

A comment stats with `--␣` or `**␣` and ends at the end of line. As a commenting convention `--␣` is used for commenting on the following CafeOBJ text and `**␣` is used for the preceding text.

The key word `mod` indicates a module declaration with a module name followed by a module body. A module body starts with { and ends with }.

The character `[` starts a declaration of sorts, i.e. a sequence of sort names, and the character `]` ends the declaration. The character `<` in the sequence of sort names indicates that any sort on the left is included in (i.e. subsort of) any sort on the right.

`ops` starts a declaration of operator names with the same rank (a pair of arity and co-arity), and `rs, ws, cs` are declared to be constants (i.e. operators without arguments) of sort `LabelLt`. `constr` is an operator attribute and indicates `rs, ws, cs` are constructors.

If names satisfy the condition that different names denote different objects, they are called "literals". `eq` starts a declaration of an unconditional equation and should end with "␣.". The equation in the module `LABEL` specify that the elements of sort `LabelLt` are literals. `_=_` in the left-hand side of the equation and `_==_` in the right-hand side are the built-in equality predicates declared for each sort. `_==_` returns `false` if the left and the right are different terms after the reduction, but `_=_` does not.

The last character `!` of `mod!` for the `LABLE` module indicates that this module denotes the standard model (the initial model), and the module `LABEL` denotes the three points set {`rs, ws, cs`} of the label literals (i.e. `LabelLt`).

The last character `*` of `mod*` for the `AID` module indicates that this module denotes any model that satisfy `AID`, and the module `AID` denotes any set of agent identifiers.

3.2 QUEUE[5], AID-QUEUE

The following `AID-QUEUE` module defines the global queue of agent identifiers.

```
-- mod* TRIV {[Elt]} is a bulit-in module
-- generic (or parameterized) queue (first in first out storage)
mod! QUEUE (X :: TRIV) {
** (X :: TRIV) declares the parameter module X should be an instance
** of the module TRIV
```

[5] The module `QUEUE` is in the file `natQuSet.cafe` on the web page.

```
-- elements and their queues, Elt comes from (X :: TRIV)
[Elt.X < Qu]
** an element in Elt is an element in Qu
-- declaration of a constructor constant
op empQ : -> Qu {constr}   ** empty queue
-- assoicative queue constructors with id: empQ
op _&_ : Qu Qu -> Qu {constr assoc id: empQ}
-- equality _=_ over the sort Qu
-- _=_ is defined for each sort in the built-in module EQL
-- and EQL is imported to each module automatically
eq (empQ = (E:Elt & Q:Qu)) = false .
ceq ((E1:Elt & Q1:Qu) = (E2:Elt & Q2:Qu)) = ((E1 = E2) and (Q1 = Q2))
    if not((Q1 = empQ) and (Q2 = empQ)) .
}
** parameterized module QUEUE just defines generic sequences
-- Queues of Aid (agent identifiers)
mod! AID-QUEUE {pr(QUEUE(AID{sort Elt -> Aid}))}
** AID-QUEUE is defined by instantiating formal parameter X of QUEUE
** with AID by viewing sort Elt as sort Aid
```

`pr(_)` indicates a **protecting** importation, and declares to import a module without changing its models.

`QUEUE(AID{sort Elt -> Aid})` defines the module obtained by instantiating the formal parameter `X` of `QUEUE` by `AID` with the interpretation of `Elt` as `Aid`.

3.3 AOB, SET[6], STATE

A QLOCK state is defined as a pair of a queue of agent identifiers and a set of agent observers by the following module `STATE`

```
-- agent observer
mod! AOB {pr(LABEL + AID)
** pr(LABEL + AID) is same as 'pr(LABEL) pr(AID)'
-- agent observer (Aob) and its constructor
[Aob] op (lb[_]:_) : Aid Label -> Aob {constr}
** (lb[A:Aid]: L:Label) is a term of the sort Aob
** that indicates an agent A is with a label L
}
-- generic sets
mod! SET(X :: TRIV) { [Elt.X < Set]
-- empty set
op empty : -> Set {constr}
-- assicative and commutative set constructor with identity
op _ _ : Set Set -> Set {constr assoc comm id: empty}
-- '_ _' is idempotent with respect to the sort Elt
eq E:Elt E S:Set = E S .
}
```

[6] The module `SET` is in the file `natQuSet.cafe` on the web page.

```
-- 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}
** the sort State has no subsort
}
```

`SET(AOB{sort Elt -> Aob})` defines sets of agent observers (i.e. `Aob`s). `*{sort Set -> State}` defines the renaming of `Set` to `State`. A state (i.e. an element of the sort `State`) is presented as a pair of a `Q:Qu` and a set of the terms of the pattern `(lb[A:Aid]: L:Label)`, where the term `(lb[A:Aid]: L:Label)` denotes that an agent `A` is in the status `L`.

Let `STATE`n denote `STATE` with n agent identifiers (i.e. `Aid` $= \{\mathtt{a}_1, \cdots, \mathtt{a}_n\}$), and let $\Sigma_{\mathtt{STATE}n}$ be the signature of `STATE`n, then `State` $= T_{\Sigma_{\mathtt{STATE}n}}$ (see Section 2.1).

3.4 WT, TY, EXc, QLOCKsys1

The QLOCK protocol is defined by the following three modules `WT, TY, EXc`. 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`, `EXc` formulate the three actions explained in Section 2.11 precisely and succinctly. `QLOCKsys1` 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)) . }
-- exc: exit transition with a condition
mod! EXc {pr(STATE)
ctrans[exc]:   ((A1:Aid & Q:Qu) $ ((lb[A2:Aid]: cs) AS:Aobs))
            => (          Q     $ ((lb[A2    ]: rs) AS)) if (A1 = A2) . }
-- system specification of QLOCK
mod! QLOCKsys1{pr(WT + TY + EXc)}
```

An unconditional transition rule starts with `trans`, contains the rule's name `[_]:`, a current state term, `=>`, a next state term, and should end with "␣.". A conditional transition rule starts with `ctrans`, contains same components as `trans`, and `if` followed by a condition (a predicate) before "␣.".

Note that the term `Q:Qu` matches any term of the sort `Qu` and the term `(lb[A:Aid]: rs)` matchs any term `(lb[aid]: rs)` with `aid` of the sort `Aid`. Note also that the second component of a state configuration is a set (i.e. a term

composed of associative, commutative, and idempotent binary constructors "_ _"). These imply that the left-hand side of the the transition rule `wt` matches to a state multiple ways depending on how many agents with `rs` are in the state, and unbounded number of transitions may be defined by the rule `wt`. The rules `ty` and `exc` have a similar nature.

For `STATE`n with `Aid` $= \{\mathtt{a}_1, \cdots, \mathtt{a}_n\}$, the transition rules `wt,ty,exc` define the one step transition relations $\rightarrow_{\mathtt{wt}}, \rightarrow_{\mathtt{ty}}, \rightarrow_{\mathtt{exc}}$ respectively on the state space `State` $= T_{\Sigma_{\mathtt{STATE}n}}$. `QLOCKsys1`$n$ with `STATE`n defines a set of transitions $Tr_{\mathtt{QLOCKsys1}n} \stackrel{\text{def}}{=} (\rightarrow_{\mathtt{wt}} \cup \rightarrow_{\mathtt{ty}} \cup \rightarrow_{\mathtt{exc}}) \subseteq (T_{\Sigma_{\mathtt{STATE}n}} \times T_{\Sigma_{\mathtt{STATE}n}})$ (see Section 2.2).

It is easily seen that the rule `ty` can be translated to a conditional rule, and the rule `exc` can be translated to an unconditional rule.[7]

4 Generic Proof Scores for Generate & Check Method

This section presents the seven parameterized CafeOBJ modules that codify the seven sufficient verification conditions of the generate & check method.[8] The seven verification conditions are the three conditions of Section 2.8 for invariant properties and the four conditions of Section 2.10 for (p leads-to q) properties.

The seven parameterized modules specifies the seven sufficient conditions in an executable way, and only by substituting the formal parameters of the parameterized modules with the specification modules of a specific system, the proof scores are completed. That is, the verifications can be done only by executing (i.e. by doing deduction via rewriting) the proof scores.

4.1 `GENcases`: Generating Patterns and Checking on Them

The module `GENcases`[9] specifies the pattern generation and the validity checking of predicates on the generated patterns.

[`check_, SqSqTr`]: The function `check_` is specified as follows and performs the validity checks on the patterns defined by `SST`. If all the validity checks are successful, `mmi(SST)` disappears and `check(SST)` returns `($):Ind`.

```
op check_ : SqSqTr -> IndTr .  eq check(SST:SqSqTr) = ($ | mmi(SST)) .
```

The sorts `SqSqTr` is specified as follows, and an `SqSqTr` (i.e. an element of the sort `SqSqTr`) is (1) an `SqSqEn` or (2) a tree (or a sequence) of `SqSqEn`s (i.e. elements of the sort `SqSqEn`) composed of the associative binary operator `_||_`. An `SqSqEn` is an `SqSq` enclosed with `[` and `]`.

[7] The file `qlock-sys-ex.cafe` on the web page contains the translated `tyc` and `ex` rules.

[8] The file `genCheck.cafe` on the web page contains the seven parameterized modules. The files `genCases.cafe`, `exState.cafe`, and `predCj.cafe` are used in `genCheck.cafe`. Note that each file without suffix "`qlock-`" in its name is not depend on the QLOCK problem and generic for the generate & check method.

[9] The module `GENcases` is in the file `genCases.cafe` on the web page page.

```
[SqSqEn < SqSqTr]
op [_] : SqSq -> SqSqEn .  op _||_ : SqSqTr SqSqTr -> SqSqTr {assoc}
```

The sort `SqSq` is specified as follows, and an `SqSq` is (1) a `ValSq`, (2) a `VlSq`, or (3) a sequence of `ValSq`s or `VlSq`s composed of the associative binary operator `_,_`that has `empSS` as an identity (`id:`). A `ValSq` is (1) a `Val` or (2) a sequence of `Val`s composed of the associative binary operator `_,_`. A `VlSq` is (1) a `Val` or (2) a sequence of `VlSq`s composed of the associative binary operator `_;_`. Note that the operator `_,_` is overloaded (i.e. denotes two different operations), and a term composed of an associative binary operator inductively is called a sequence for `SqSq`, `ValSq`, `VlSq` while a `SqSqTr` is called a tree.

```
[Val < ValSq] op _,_ : ValSq ValSq -> ValSq {assoc}
[Val < VlSq] op _;_ : VlSq VlSq -> VlSq {assoc}
[ValSq VlSq < SqSq]
op empSS : -> SqSq . op _,_ : SqSq SqSq -> SqSq {assoc id: empSS}
```

[**Alternative Expansion with** `_;_`]**:** The operator `_;_` specifies possible alternatives and the following equation expands alternatives `_;_` into a term composed of the operator `_||_`.

```
eq [(SS1:SqSq,(V:Val;VS:VlSq),SS2:SqSq)]
   = [(SS1,V,SS2)] || [(SS1,VS,SS2)] .
```

The equation applies recursively and any subterm with alternatives `_;_` is expanded into a term with `_||_`. It implies that for any term *sqSq* of the sort `SqSq` the term `[`*sqSq*`]` is reduced to the term composed by applying the operator `_||_` to terms of the form `[`$valSq_i$`]` (i = 1,2,$\cdots$) for $valSq_i$ of the sort `ValSq`. For example, if terms `v1`, `v2`, `v3` are of the sort `Val`, the following reduction happens. Note that, because `empSS` is declared to be an identity for the operator "`_,_ : SqSq SqSq -> SqSq`", the equation covers the cases in which `SS1` and/or `SS2` in the left-hand side of the equation are/is `empSS`.

```
[(v1;v2;v3),(v1;v2)]
=red=>
[ (v1 , v1) ] || [ (v2 , v1) ] || [ (v3 , v1) ] ||
[ (v1 , v2) ] || [ (v2 , v2) ] || [ (v3 , v2) ]
```

To make the alternative expansion with `_;_` more versatile, the functions `t__` and `g__` are introduced as follows. `String` is a sort from the CafeOBJ built-in module `STRING` and denotes the set of character strings like `"abc"`, `"v1"`, `"_%_"`. By using `t__`, a user is supposed to specify term constructors with appropriate identifiers in the first argument, and accompanying `g__` can be used to specify the alternative expansion with `_;_` and the constructors. The two equations for `g__` make the expansion of a nested expression with `[_]`s and `_;_`s possible, and reduce "`g` *st sqSqTr*" to "`t` *st sqSqTr*" if *sqSqTr*" is of the sort `ValSq`.

```
op t__ : String ValSq -> Val .
op g__ : String SqSqTr -> VlSq .
eq g(S:String)(SST1:SqSqTr || SST2:SqSqTr) = (g(S) SST1);(g(S) SST2) .
eq g(S:String)[VSQ:ValSq] = t(S)(VSQ) .
```

For an example, let the following equations for `t__` be given.[10]

```
[Qu Aid Label Aobs State < Val]
eq t("lb[_]:__")(A:Aid,L:Label,AS:Aobs) = ((lb[A]: L) AS) .
eq t("_$_")(Q:Qu,AS:Aobs) = (Q $ AS) .
```

Then the following expansion by reduction of alternatives is possible for QLOCK state terms if we assume `q` is of the sort `Qu`, `a1` and `a2` are of the sort `Aid`, and `as` is of the sort `Abos`.

```
[(g("_$_")[(empQ;(a1 & q)),(g("lb[_]:__")[a2,(rs;ws;cs),as])])]
=red=>
[(empQ $ ((lb[a2]: rs) as))] || [((a1 & q) $ ((lb[a2]: rs) as))] ||
[(empQ $ ((lb[a2]: ws) as))] || [((a1 & q) $ ((lb[a2]: ws) as))] ||
[(empQ $ ((lb[a2]: cs) as))] || [((a1 & q) $ ((lb[a2]: cs) as))]
```

The specifications of alternative expansions with `_;_`, `[_]`, `g__` are called **alternative scripts** or **alternative expansion scripts**. Alternative scripts are simple but powerful enough to specify a fairly large number of necessary patterns. Note that an alternative script is a term of the sort `SqSqTr`.

[`IndTr`, `mmi_`, `mi_`, `v_`]: The sort `IndTr` and the function `mmi_` are specified as follows, and `mmi_` translates a `SqSqTr` to a `IndTr` and `mmi[`*sqSq*`]` reduces to `mi(`*sqSq*`)` if *sqSq* is of the sort `ValSq`.

```
-- indicator and indicator tree
[Ind < IndTr]
op $ : -> Ind .
op _|_ : IndTr IndTr -> IndTr {assoc}
-- make make indicator
op mmi_ : SqSqTr -> IndTr .
eq mmi(SST1:SqSqTr || SST2:SqSqTr) = (mmi SST1) | (mmi SST2) .
eq mmi[VSQ:ValSq] = mi(VSQ) .
```

The indicator `i__` and the making indicator function `mi_` are specified as follows. The functions `ii` (information indicator) and the predicate `v_` to be checked on `ValSq` are supposed to be defined by a user. `mi(`*valSq*`)` reduces to "`(i v(`*valSq*`) ii(`*valSq*`))`", and disappears if the first argument `v(`*valSq*`)` reduces to `true`. This implies that the predicate `v_` is valid for all the `ValSq`s specified by `SST` if `check(SST)` returns `($):Ind`.

```
-- indicator
[Info] op i__ : Bool Info -> Ind .
-- making any indicator with 'true' disappear
eq (i true II:Info) | IT:IndTr = IT .
eq IT:IndTr | (i true II:Info) = IT .
-- information constructor
op ii_ : ValSq -> Info .
-- the predicate to be checked
pred v_ : ValSq .
-- make indicator for v_
op mi_ : ValSq -> Ind .
eq mi(VSQ:ValSq) = (i v(VSQ) ii(VSQ)) .
```

[10] These equations are in the file `qlock-genStTerm.cafe` on the web page page.

4.2 Three Parameterized Modules for Invariant Properties

[PREDcj]: For defining conjunctions of predicates flexibly, the following parameterized module PREDcj[11] is prepared.

```
-- defining the conjunction of predicates
-- via the sequence of the names of the predicates
mod! PREDcj (X :: TRIV) {
-- names of predicates on Elt.X and the sequences of the names
[Pname < PnameSeq]
-- associative binary operator for constructing non nil sequences
op _ _ : PnameSeq PnameSeq -> PnameSeq {constr assoc}
-- cj(pns,e) defines the conjunction of predicates
-- whose names constitute the sequence pns
op cj : PnameSeq Elt -> Bool .
eq cj((PN:Pname PNS:PnameSeq),E:Elt) = cj(PN,E) and cj(PNS,E) . }
```

By using the cj (conjunction) operator of PREDcj, a conjunction of predicates can be expressed just as a sequence of the names of the predicates. This helps prompt modifications of component predicates of *inv* in the checks of the conditions (1),(2),(3) of Section 2.8 and the conditions (1),(2),(3),(4) of Section 2.10.

[INV-1v, INV-2v[12]]: The following two parameterized modules INV-1v and INV-2v codify the verification conditions (1) and (2) of Section 2.8 directory. The theory module STEpcj specifies the modules with sorts corresponding to Ste, Pname, PnameSeq and a function corresponding to cj that make a predicate be presented as cj(*pNameSeq*,*ste*).

By defining the predicate v_ of the module GENcases as the predicate $\mathtt{p_{st\text{-}1}}$ of the condition (1) or the predicate $\mathtt{p_{st\text{-}2}}$ of the condition (2), necessary checks are done on all state patterns. The PnameSeqs p-iinv (two), p^t, and p-init are supposed to be reified after the parameter modules are substituted with actual specification modules (i.e. after the instantiation of parameter modules).

```
mod* STEpcj {[Ste] [Pname < PnameSeq] pred cj : PnameSeq Ste .}
mod! INV-1v (ST :: STEpcj) {ex(GENcases)
  -- possible inductive invariant and target predicate
  ops p-iinv p^t : -> PnmSeq .
  [Ste < Val] eq v(S:Ste) = cj(p-iinv,S:Ste) implies cj(p^t,S) . }
mod! INV-2v (ST :: STEpcj) {ex(GENcases)
  ops p-init p-iinv : -> PnmSeq .
  [Ste < Val] eq v(S:Ste) = cj(p-init,S) implies cj(p-iinv,S) . }
```

[VALIDq, G&C-Tv, INV-3q[13]]: The following parameterized module VALIDq directly specifies valid-q of Section 2.7. inc(RWL) declares the importation of the built-in module RWL that is necessary for using the built-in search predicate.

[11] The the module PREDcj is in the file predCj.cafe on the web page.
[12] The modules INV-1v, INV-2v are in the file genCheck.cafe on the web page.
[13] The modules VALIDq, G&C-Tv, INV-3q are in the file genCheck.cafe on the web page.

```
mod* STE {[Ste]}
mod! VALIDq (X :: STE) {inc(RWL)
  -- predicate to be checked for all the transitions
  pred q : Ste Ste .
  -- information constructor
  [Infom] op (ifm _ _ _ _) : Ste Ste Bool Bool -> Infom {constr}
  pred _then _ : Bool Bool .
  eq (true  then B:Bool) = B .  eq (false then B:Bool) = true .
  pred valid-q : Ste Ste Bool .
  eq valid-q(S:Ste,SS:Ste,CC:Bool) =
     not(S =(*,1)=>+ SS if CC suchThat
         not((CC then q(S, SS)) == true) {(ifm S SS CC q(S,SS))}) . }
```

The following module G&C-Tv defines v(S:Ste,SS:Ste,CC:Bool) as valid-q(S, SS,CC). Note that S:Ste,SS:Ste,CC:Bool in the left-hand side is of the sort ValSq but S,SS,CC in the right-hand side is of the sort Ste,Ste,Bool that is the sort list (or arity) of the standard form (i.e. without _) operator valid-q. In the module INV-3q, by defining q of the module VALIDq as q_{tr-3} of the condition (3) of Section 2.8, necessary checks are done on all state patterns. The PnameSeq p-iinv is supposed to be reified after the instantiation of the parameter module "ST :: STEpcj".

```
mod! G&C-Tv (S :: STE) {ex(VALIDq(S) + GENcases)
  [Ste Bool < Val] eq v(S:Ste,SS:Ste,CC:Bool) = valid-q(S,SS,CC) . }
mod! INV-3q (ST :: STEpcj) {ex(G&C-Tv(ST))
  op p-iinv : -> PnmSeq .
  eq q(S:Ste,SS:Ste) = (cj(p-iinv,S) implies cj(p-iinv,SS)) . }
```

Note that the three parameterized modules INV-1v, INV-2v, INV-3q have the same parameter declaration "(ST :: STEpcj)". It indicates that the modules obtained by applying the parameterized module PREDcj to appropriate modules can be substituted for the parameter modules of these three parameterized modules.

4.3 Four Parameterized Modules for (p leads-to q) Properties

[EX-STATE, PCJ-EX-STATE[14]]: For specifying the four verification conditions for (p leads-to q) properties, the states are needed to extend with data. The following parameterized module EX-STATE specifies the state extension following Lemma 10 directly. The theory module ST-DT requires functions p, q, m for (p leads-to q) properties, and cj for defining predicates via their names. The functions p, q, m on ExState are specified based on the functions p, q, m on State and Data. The transitions over ExState are specified based on the transitions over State by declaring two equations with the built-in search predicates _=(*,1)=>+_if_suchThat_{_} and _=(*,1)=>+_. The equation for t__ is for composing a term of the sort ExState with the constructor _%_ in the alternative expansion script.

[14] The modules EX-STATE, PCJ-EX-STATE are in the file exState.cafe on the web page.

```
-- theory module with state and data
mod* ST-DT {ex(PNAT)
  [Ste Data] ops p q : Ste Data -> Bool . op m : Ste Data -> Nat.PNAT .
  [Pnm < PnmSeq] op cj : PnmSeq Ste -> Bool . }
mod! EX-STATE (SD :: ST-DT) {inc(RWL) ex(GENcases)
  [ExState Infom]
  -- state constructor for extended states
  op _%_ : Ste Data -> ExState {constr}
  -- the transitions on ExState is the same as the transitons on Ste
  eq ((S:Ste % D:Data) =(*,1)=>+ (SS:Ste % D)
      if CC:Bool suchThat B:Bool {I:Infom})
     = (S =(*,1)=>+ SS if CC suchThat B {I}) .
  eq ((S:Ste % D:Data) =(*,1)=>+ (SS:Ste % D)) = (S =(*,1)=>+ SS) .
  -- predicates p and q on ExState
  ops p q : ExState -> Bool .
  eq p(S:Ste % D:Data) = p(S,D) .  eq q(S:Ste % D:Data) = q(S,D) .
  -- measure function on ExState
  op m : ExState -> Nat.PNAT . eq m(S:Ste % D:Data) = m(S,D) .
  -- t__  is introduced in the module GENcases
  [Ste Data ExState < Val] eq t("_%_")(S:Ste,D:Data) = (S % D) . }
```

The following parameterized module `PCJ-EX-STATE` makes the `cj` available on `ExState` and relate that to the `cj` on `Ste`.

```
mod! PCJ-EX-STATE (SD :: ST-DT) {
  ex((PREDcj((EX-STATE(SD)){sort Elt -> ExState}))
     *{sort Pname -> ExPname, sort PnameSeq -> ExPnameSeq})
  [Pnm < ExPname] [PnmSeq < ExPnameSeq]
  eq cj(PN:Pnm,(S:Ste % D:Data)) = cj(PN,S) . }
```

[`PQ-1q`, `PQ-2q`, `PQ-3v`, `PQ-4v`[15]]: The four parameterized modules for the four verification conditions for (p leads-to q) properties are specified as follows. These are direct translation from the four conditions of Section 2.10. The parameterized modules `PQ-1q` and `PQ-2q` are using Generate&Check-T1 or Generate&Check-T2, and the parameterized module `G&C-Tv` is necessary for reifying the predicate `q`. The parameterized modules `PQ-3v`, `PQ-4v` are using Generate&Check-S, and only the module `GENcases` is necessary for reifying the predicate `v_`.

```
-- theory module with p,q,m,cj on states
mod* STPQpcj {ex(PNAT)
  [Ste] ops p q : Ste -> Bool . op m : Ste -> Nat.PNAT .
  [Pnm < PnmSeq] op cj : PnmSeq Ste -> Bool . }
mod! PQ-1q (SQ :: STPQpcj) {ex(G&C-Tv(SQ))
  op pq-1-inv : -> PnmSeq .
  eq q(S:Ste,SS:Ste) =
     (cj(pq-1-inv,S) and p(S) and not(q(S))) implies (p(SS) or q(SS)) . }
mod! PQ-2q (SQ :: STPQpcj) {ex(G&C-Tv(SQ))
  op pq-2-inv : -> PnmSeq .
```

[15] The modules `PQ-1q`, `PQ-2q`, `PQ-3v`, `PQ-4v` are in the file `genCheck.cafe` on the web page.

```
  eq q(S:Ste,SS:Ste) =
     (cj(pq-2-inv,S) and p(S) and not(q(S))) implies (m(S) > m(SS)) . }
mod! PQ-3v (SQ :: STPQpcj) {inc(RWL) ex(GENcases)
  op pq-3-inv : -> PnmSeq .  [Ste < Val]
  eq v(S:Ste,SS:Ste) =
     (cj(pq-3-inv,S) and p(S) and not(q(S))) implies (S =(*,1)=>+ SS) . }
mod! PQ-4v (SQ :: STPQpcj) {pr(GENcases)
  op pq-4-inv : -> PnmSeq . [Ste < Val]
  eq v(S:Ste) =
     (cj(pq-4-inv,S) and (p(S) or q(S)) and (m(S) = 0)) implies q(S) . }
```

Note that the four parameterized modules `PQ-1q`, `PQ-2q`, `PQ-3v`, `PQ-4v` have the same parameter declaration "`(SQ :: STPQpcj)`". It indicates that the modules obtained by applying the parameterized module `PCJ-EX-STATE` to appropriate modules can be substituted for the parameter modules of these four parameterized modules.

5 Property Specifications of QLOCK[16]

The property specifications for the generate & check method are supposed to specify the following predicates.

1. The possible inductive invariant predicate inv, the target state predicate p^t, and the initial state predicate $init$ for verifying invariant properties (Section 2.8).
2. The invariant predicate inv, the predicates p, q and the measure function m for verifying (p leads-to q) properties (Section 2.10). There are four verification conditions for each (p leads-to q) property, and the invariant predicate inv may be different depending on the conditions.

Usually, the predicates are specified as conjunctions of elemental predicates.

For QLOCK, we adopt a strategy of formalizing necessary functions and elemental predicates based on the Peano style natural numbers. The strategy works well especially for specifying the measure function m for a (p leads-to q) property as demonstrated in Section **??**.

5.1 Basic Functions on State

The module `PNAT`[17] defines Peano style unary natural numbers `Nat` with the two constructors "`op 0 : -> Nat {constr}`", "`op s_ : Nat -> Nat {constr}`", and two functions `_+_` (addition), `_>_` (greater than).

Based on `PNAT`, the module `STATEfuns` defines fundamental functions on states which are necessary to specify necessary elemental predicates. The functions are to get "the queue in a state" (`qu`), "the agent observations in a state" (`aos`),

[16] The file `qlock-prop.cafe` on the web page contains the property specifications of QLOCK.

[17] The module `PNAT` is in the file `natQuSet.cafe` on the web page.

"length of an `Aobs`" (`#laos`), "the number of a label in an `Aobs`" (`#lss`), "the number of a label in a state" (`#ls`), "the number of an aid in an `Aobs`" (`#ass`), "the number of an aid in a state" (`#as`), "the number of an aid in a state" (`#as`), "the number of an aid in a queue" (`#aq`), "label of an agent in an `Aobs`" (`laga`), "label of an agent in a `State`" (`lags`). Note that an expression like "an `Aobs`" means "an element of the sort `Aobs`", e.g. "a state" is same as "a `State`". All of these functions are easily defined with recursive equations. For example `#lss` and `laga` are specified as follows.

```
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 .
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 .
```

5.2 Initial State Predicate

Based on `STATEfuns`, the elemental predicates on `State` for defining initial states are specified in the module `STATEpred-init`. An initial state predicate (i.e. a predicate to characterize the initial states) is specified in the module `INIT` as the conjunction of the elemental predicates.

By applying the parameterized module `PREDcj` to the module `STATE`, the module `STATEpcj` is obtained. Then `STATEpcj` is used in the module `STATEpred-init` for defining the following predicate via their names. "at least one agent in a state" (`aoa`), "no duplication of an `Aid` in a Aobs" (`1as`), "no duplication of an `Aid` in a state" (`1a`), "the queue is empty" (`qe`), "any `Aid` is in `rs` status" (`allRs`). The predicates specified with `aoa` and `1a` are structural requirements for a state to be well formed, and the conjunction of `aoa` and `1a` is named `wfs` (well formed state). The following shows `STATEpcj` and `STATEpred-init` (`aoa`, `1as`, `1a`, `qe` are omitted).

```
-- PREDcj with STATE for 'X :: TRIVE' by viewing Elt as State
mod! STATEpcj {pr(PREDcj(STATE{sort Elt -> State}))}
mod! STATEpred-init {ex(STATEfuns + STATEpcj)
  ...
  op wfs : -> Pname .  eq[wfs]: wfs = aoa 1a .
  ...
  op allRs : -> Pname .
  eq[allRs]: cj(allRs,S:State) = (#ls(S,ws)= 0) and (#ls(S,cs)= 0) . }
```

By using predicate names specified in `STATEpred-init`, the initial state predicate *init* is specified with the predicate name sequence `init` in the following module `INIT`.

```
mod! INIT {ex(STATEpred-init)
  op init : -> PnameSeq .  eq init = wfs qe allRs . }
```

5.3 Elemental Invariant Predicates and Target Predicate

The elemental invariant predicates are supposed to be elements for composing inductive invariant predicates as conjunctions. The target predicate p^t is usually chosen as one of the elemental predicates, and the first condition of the invariant verification (Section 2.8) is trivial.

The module `AID-QUEUE-a` defines queues with head (`hd`) and tail (`tl`) operators and used by the module `STATEpred-inv` that defines the elemental predicates necessary for defining inductive invariants. `STATEpred-inv` defines the predicates for "mutual exclusion property" (`mx`), and for the situations "if queue is empty" (`qep`), "if agent is in rs" (`rs`), "if agent is in ws" (`ws`), "if agent is in cs" (`cs`).

The property `mx` means that at most one agent is with `cs` at any time, and it is the target property p^t to be verified.

`mx`, `qep`, and `rs` are specified as follows.

```
op mx : -> Pname .
eq[mx]: cj(mx,S:State) = (#ls(S,cs) = 0) or (#ls(S,cs) = (s 0)) .
ops qep rs : -> Pname .
eq[qep]: cj(qep,(Q:Qu $ ((lb[A:Aid]: L:Label) AS:Aobs)))
         = ((Q = empQ) implies (#lss(((lb[A]: L) AS),cs) = 0)) .
eq[:m-and rs]: cj(rs,(Q:Qu $ ((lb[A:Aid]: L:Label) AS:Aobs)))
               = ((L = rs) implies (#aq(Q,A) = 0)) .
```

An equation with the `:m-and` (matching and) attribute like "`eq[:m-and` id`]:` l `=` r ." indicates that (1) left-hand side l and right-hand side r of the equation are of the sort `Bool`, and (2) if there are multiple matches $m_1, \cdots, m_k (k = 2, 3, \cdots)$ (match m_i is a map from variables to terms and is extended to a map from terms to terms) such that $m_i(l) = t$ for a Boolean term t, the equation "$l = r$" is defined to be instantiated to the equation "t `=` $m_1(r)$ `and-also` $m_2(r)$ `and-also` $\cdots$ `and-also` $m_k(r)$" (`_and-also_` is a variant of the built-in operator `_and_` on `Bool`).

Hence, by matching the left-hand side of the equation `rs` (i.e. "`eq[:m-and rs]: ...`") with a Boolean term "`cj(rs, (q $ ((lb[a1]: l1) (lb[a2]: l2) as)))`", we get the following instantiated equation. Note that "`((l1 = rs) implies (#aq (q,a1) = 0))`" is gotten with the match {`Q`→`q`, `A`→`a1`, `L`→`l1`, `AS`→`((lb[a2]: l2) as)`} and "`((l2 = rs) implies (#aq(q, a2) = 0))`" is gotten with the match {`Q`→`q`, `A`→`a2`, `L`→`l2`, `AS`→`((lb[a1]: l1) as)`}.

```
cj(rs,(q $ ((lb[a1]: l1) (lb[a2]: l2) as)))
= ((l1 = rs) implies (#aq(q,a1) = 0)) and-also
  ((l2 = rs) implies (#aq(q,a2) = 0)) .
```

The above equation is the correct interpretation of the equation `rs` because an `Aobs` is a set of elements of the sort `Aob` and the pattern `(lb[A:Aid]: L:Label)` of the sort `Aob` in the left-hand side of the equation `rs` should match each element of an `Aobs` conjunctively. The `:m-and` attribute of the CafeOBJ language realizes this correct interpretation.

5.4 Predicates and Measure Function for (p leads-to q) property

The following module `PQMonState` specifies the `p` and `q` predicates and a measure function `m` for defining a (p leads-to q) property of QLOCK. The function `lags`

denotes the label of an agent in a state (see the module `STATEfuns`). The function `#dms` is specified in the following module `STATEfuns-pq` and intended to denote the properly decreasing natural numbers according to the state transitions of QLOCK. `PNAT*`[18] is `PNAT` with the `_*_` (times) operation.

```
mod! STATEfuns-pq {ex(STATEfuns + PNAT*)
  -- 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 (p leads-to q) property
  op #dms : State Aid -> Nat .
  eq #dms(S:State,A:Aid)
     = ((s s s 0) * #daq(qu(S),A)) + #ls(S,rs) + #ccs(S) . }
mod! PQMonState {ex(STATEpcj + STATEfuns-pq)
  ops p q : State Aid -> Bool . eq p(S:State,A:Aid) = (lags(S,A) = ws) .
                                eq q(S:State,A:Aid) = (lags(S,A) = cs) .
  op m : State Aid -> Nat.PNAT . eq m(S:State,A:Aid) = #dms(S,A) . }
```

The (`q` leads-to `q`) property of QLOCK is verified in Section **??** .

5.5 Extended State `(State % Aid)` and Inductive Invariants

For using Generate&Check-S or Generate&Check-T1/T2 with `p(S:State,A:Aid)`, `q(S:State,A:Aid)`, and `m(S:State,A:Aid)` of the module `PQMonState`, `State` needs to be extended with `Aid`. The following module `EX-PQMonST` extends `State` of `PQMonState` with `Aid` for constructing `ExState` (= `(State % Aid)`) by applying the parameterized module `PCJ-EX-STATE` to the module `PQMonState`.

```
-- ExState with PQMonST/State and Aid/Date
-- and a new invariant qas on ExState = (State % Date)
mod! EX-PQMonST {
  pr((PCJ-EX-STATE(
        PQMonST{sort Ste -> State, sort Data -> Aid,
                sort Pnm -> Pname, sort PnmSeq -> PnameSeq})))
  -- if an agent is in the Qu then the agent is in Aobs
  op qas : -> ExPname .
  eq cj(qas,((Q:Qu $ AS:Aobs) % A:Aid))
     = ((#aq(Q, A) = s 0) implies not(#ass(AS, A) = 0)) . }
```

A new elemental predicate is defined with the name `qas` of the sort `ExPname` in `EX-PQMonST`. By using the predicate names defined in `STATEpred-inv` and `qas`, possible inductive invariants are specified in the following module `INV`.

```
-- possible inductive invariant predicates
mod! INV {ex(EX-PQMonST + STATEpred-inv)
  ops inv1 inv2 : -> PnameSeq .  ops inv3 : -> ExPnameSeq .
  eq inv1 = wfs .  eq inv2 = mx qep rs ws cs . eq inv3 = qas . }
```

[18] The module `PNAT*` is in the file `natQuSet.cafe` on the web page.

In Section **??**, the conjunction "`inv1 inv2 inv3`" is proved to be an inductive invariant. As a matter of fact, any of `inv1`, `inv2`, or `inv3` can be proved to be an inductive invariant.

6 Proof Scores for QLOCK

Proof scores for QLOCK can be obtained by substituting the parameter modules of the seven parameterized modules in Section 4.2 and 4.3 with appropriate QLOCK specification modules.

The three modules `Q-INV-1v`, `Q-INV-2v`, `Q-INV-3q` for the three invariant verification conditions and the four modules `Q-PQ-1q`, `Q-PQ-2q`, `Q-PQ-3v`, `Q-PQ-4v`[19] for the four (p leads-to q) property verification conditions of QLOCK are obtained by substituting the parameter modules of the parameterized modules `INV-1v`, `INV-2v`, `INV-3q`, and `PQ-1q`, `PQ-2q`, `PQ-3v`, `PQ-4v` with the module `EX-PQMonST`.

6.1 Proof Scores for invariant properties

The following proof scores prove the three verification conditions for invariant properties under the condition in which each predicate inv, p^t, or $init$ is defined with the predicate name sequence "`inv1 inv2 inv3`", `mx`, or `init` respectively.

[`Q-INV-1-genCheck`[20]]: The reduction "`red ck .`" in the following proof score proves the first verification condition "`(cj(inv1 inv2 inv3,S) implies cj(mx, S))` for any state `S` of the sort `ExState`" if it returns `($):Ind`.

```
mod! Q-INV-1-genCheck {ex(Q-INV-1)
  op ck : -> IndTr . eq ck = check([‘s:ExState]) . }
open Q-INV-1-genCheck . pr(INV)
  eq p-iinv = inv1 inv2 inv3  . eq p^t = mx . red ck . close
```

`open` opens the module `Q-INV-1-genCheck`; `pr(INV)` imports a necessary module `INV`; two equations reify predicate names `p-iinv` and `p^t` as "`inv1 inv2 inv3`" and `mx` respectively; "`red ck .`" reduces `check([‘s:ExState])` and returns the result, and `close` closes the opened tentative module.

`cj(inv1 inv2 inv3,S)` trivially implies `cj(mx,S)` for any `S` because `inv2` includes `mx`, hence the most general trivial state pattern `‘s:ExState` is enough.

[`Q-INV-2-genCheck`[21]]: The reduction "`red ck .`" in the following proof score proves the second verification condition "`(cj(init,S) implies cj(inv1 inv2 inv3, S))` for any state `S` of the sort `ExState`" if it returns `($):Ind`.

```
mod! Q-INV-2-genCheck {ex(Q-INV-2 + GENstTerm + CONSTandLITL)
op ck : -> IndTr .
eq ck = check( [(g("_%_")[(g("_$_")[q,empty]),a1])]  ||
```

[19] The modules `Q-INV-1v`, `Q-INV-2v`, `Q-INV-3q`, and `Q-PQ-1q`, `Q-PQ-2q`, `Q-PQ-3v`, `Q-PQ-4v` are in the file `qlock-genCheck.cafe` on the web page.

[20] The module `Q-INV-1-genCheck` is in the file `qlock-inv-1-ps.cafe` on the web.

[21] The module `Q-INV-2-genCheck` is in the file `qlock-inv-2-ps.cafe` on the web page.

```
  [(g("_%_")[(g("_$_")[empQ,(g("lb[_]:__")[a1,(rs;ws;cs),as])]),a2])] ||
  [(g("_%_")[(g("_$_")[(a1 & q),(g("lb[_]:__")[a2,(rs;ws;cs),as])]),a3])]
  ) . }
open Q-INV-2-genCheck .  pr(INIT + INV)
  eq p-init = init . eq p-iinv = inv1 inv2 inv3 .  red ck . close
```

The module `GENstTerm` is necessary to use `g__` in the alternative script (the argument of `check_`). The module `CONSTandLITL`[22] includes fresh constant declarations "`op q : -> Qu . op as : -> Aobs . ops a1 a2 a3 : -> Aid .`" that are used in the alternative script. The two equations reify the predicate names `p-init` and `p-iinv` as `init` and "`inv1 inv2 inv3`" respectively.

The alternative script expands to the term of the sort `SqSqTr` that includes the state patterns that subsume all the ground state terms of the sort `ExState`.[23] Hence, by Generate&Check-S in Section 2.6, if "`red ck .`" returns `($):Ind` then `(cj(init,S) implies cj(inv1 inv2 inv3,S))` for any state `S` of the sort `ExState` (i.e. any ground term of the sort `ExState`).

[`Q-INV-3-genCheck`[24]]: The reduction "`red ck .`" in the following proof score proves the third verification condition "`(cj(inv1 inv2 inv3,S) implies cj(inv1 inv2 inv3,SS))` for any transitions `(S,SS)` of QLOCK with `ExState` (i.e. `S` and `SS` are of the sort `ExState`)" if it returns `($):Ind`.

```
mod! Q-INV-3-genCheck {ex(Q-INV-3 + GENstTerm + CONSTandLITL)
 ops sst1 sst2 sst3 : -> SqSqTr .
 eq sst1 = [(g("_%_")[(g("_$_")[empQ,(g("lb[_]:__")[b1,rs,as])]),
                      (b1;b2)]),SS:ExState,CC:Bool]  ||
           [(g("_%_")[(g("_$_")[(b1 & q),(g("lb[_]:__")[(b1;b2),rs,as])]),
                      (b1;b2;b3)]),SS,CC] .
 eq sst2 = [(g("_%_")[(g("_$_")[(b1 & q),(g("lb[_]:__")[b1,ws,as])]),
                      (b1;b2)]),SS:ExState,CC:Bool] .
 eq sst3 = [(g("_%_")[(g("_$_")[(b1 & q),(g("lb[_]:__")[(b1;b2),cs,as])]),
                      (b1;b2;b3)]),SS:ExState,CC:Bool] .
 op ck : -> IndTr .  eq ck = check(sst1 || sst2 || sst3) . }
-- Generate&Check-T1
open Q-INV-3-genCheck . pr(QLOCKsys1 + INV + FACTtbu)
 eq p-iinv = inv1 inv2 inv3 . red ck . close
```

The alternative script `sst1` specifies three state patterns that cover the left-hand side `(Q:Qu $ ((lb[A:Aid]: rs) as:Aobs))` of the transition rule `wt. b1`, `b2`, `b3` are fresh constant literals declared in the module `CONSTandLITL`. Note that all possibilities of tree occurrences of `Aid` are covered by `(...[(...[(b1 ...),(...[(b1;b2)...])]),(b1;b2;b3)])`. Note also that `SS:State,CC:Bool` in `sst1` are necessary for using the built-in search predicate "`_=(*,1)=>+_if_suchThat_{_}`". The script `sst2` specifies the state pattern `((b1 & q) $ ((lb[b1]: ws) as))` that directly cover the left-hand side `((A:Aid & Q:Qu) $ ((lb[A]: ws) AS:Aobs))` of

[22] The module `CONSTandLITL` is in the file `qlock-constAndLitl.cafe` on the web.

[23] You can see the expanded term in the comment part after the `eof` in the file `qlock-inv-2-ps.cafe` on the web.

[24] The module `Q-INV-3-genCheck` is in the file `qlock-inv-3-ps.cafe` on the web.

the transition rule `ty`. The alternative script `sst3` specifies state patterns that cover the left-hand side `((A1:Aid & Q:Qu) $ ((lb[A2:Aid]: cs) as:Aobs))` of the transition rule `exc`. Hence, by Generate&Check-T1 in Section 2.7, the correctness of the above proof score is implied.

Note that the module `FACTtbu`[25] that declare the fundamental facts like "`eq [NatGt1]: ((N:Nat.PNAT = 0) and (N > 0)) = false .`" is necessary here. Facts declared in `FACTtbu` can be proved with another proof scores usually by using induction on term structures.

By using Generate&Check-T2 instead of Generate&Check-T1, checking the transition rules one by one is possible. For doing that, instead of importing `QLOCKsys` and reducing "`check(sst1 || sst2 || sst3)`", importing the modules `WT`, `TY`, `EXc` and reducing `check(sst1)`, `check(sst2)`, `check(sst3)` one by one is enough as follows.

```
-- Generate&Check-T2
open Q-INV-3-genCheck . pr(WT + INV + FACTtbu)
  eq p-iinv = inv1 inv2 inv3 . red check(sst1) . close
open Q-INV-3-genCheck . pr(TY + INV + FACTtbu)
  eq p-iinv = inv1 inv2 inv3 . red check(sst2) . close
open Q-INV-3-genCheck .  pr(EXc + INV + FACTtbu)
  eq p-iinv = inv1 inv2 inv3 . red check(sst3) . close
```

Using Generate&Check-T2 is an effective way for detecting errors in specifications and for finding necessary lemmas during proof score developments.

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

Let the predicates `p` and `q` be defined as in the module `PQMonState`, then QLOCK enjoys the `(p(_,A:Aid) leads-to q(_,A:Aid))` (i.e. (`(lags(_,A) = ws)` leads-to `(lags(_,A) = cs)`)) property for any agent `A`. That is, if an agent gets the `ws` label then it will surely get the `cs` label.

This section presents proof scores for the four verification conditions of the `(p(_,A:Aid) leads-to q(_,A:Aid))` property.

[`Q-PQ-1-genCheck`[26]]: The reduction "`red ck .`" in the following proof score proves the first verification condition "`(cj(pq-1-inv,S) and p(S) and not(q(S))) implies (p(SS) or q(SS))` for any transition `(S,SS)` of QLOCK with `ExState`" if it returns `($):Ind`.

```
mod! Q-PQ-1-genCheck {ex(Q-PQ-1 + GENstTerm + CONSTandLITL)
  op ck : -> IndTr .
  eq ck = check(
  [(g("_%_")[(g("_$_")[empQ,(g("lb[_]:__")[b1,rs,as])]),
             (b1;b2)]),SS:ExState,CC:Bool] ||
  [(g("_%_")[(g("_$_")[(b1 & q),(g("lb[_]:__")[(b1;b2),rs,as])]),
             (b1;b2;b3)]),SS,CC]            ||
```

[25] The module `FACTtbu` is in the file `qlock-factTbu.cafe` on the web page.
[26] The module `Q-PQ-1-genCheck` is in the file `qlock-pq-1-ps.cafe` on the web page.

```
  [(g("_%_")[(g("_$_")[(b1 & q),(g("lb[_]:__")[b1,ws,as])]),
            (b1;b2)]),SS,CC]                ||
  [(g("_%_")[(g("_$_")[(b1 & q),(g("lb[_]:__")[(b1;b2),cs,as])]),
            (b1;b2;b3)]),SS,CC] ) . }
open Q-PQ-1-genCheck . pr(QLOCKsys1) red ck . close
```

`pq-1-inv` is not reified in the `open...close` clause, and it means `(cj(pq-1-inv,S)` in the premiss of the first verification condition is not necessary.

The alternative script at the argument position of `check_` specifies state patterns of the sort `ExState`. The same discussion about the module `Q-INV-3-genCheck` can apply and the alternative script of module `Q-PQ-1-genCheck` covers the left-hand sides of the three transition rules on `ExState`. Note that the three transition rules on `ExState` is not defined directly, but induced from the three transition rules on `State` with the following two equations defined in the module `EX-STATE`.

```
eq ((S:State % D:Data) =(*,1)=>+ (SS:State % D)
     if CC:Bool suchThat B:Bool {I:Infom})
   = (S =(*,1)=>+ SS if CC suchThat B {I}) .
eq ((S:State % D:Data) =(*,1)=>+ (SS:State % D)) = (S =(*,1)=>+ SS) .
```

Hence, by Generate&Check-T1 in Section 2.7, the correctness of the above proof score is implied.

[`Q-PQ-2-genCheck`[27]]: The two reductions "`red ck .`" in the following proof score prove the second verification condition "`(cj(pq-2-inv,S) and p(S) and not (q(S))) implies (m(S) > m(SS))` for any transition `(S,SS)` of QLOCK with `ExState`" if each of them returns `($):Ind`.

```
mod! Q-PQ-2-genCheck {ex(Q-PQ-2 + GENstTerm + CONSTandLITL)
  op ck : -> IndTr .
  eq ck = check(
  [(g("_%_")[(g("_$_")[empQ,(g("lb[_]:__")[b1,rs,as])]),
             (b1;b2)]),SS:ExState,CC:Bool] ||
  [(g("_%_")[(g("_$_")[(b1 & q),(g("lb[_]:__")[(b1;b2),rs,as])]),
             (b1;b2;b3)]),SS,CC]           ||
  [(g("_%_")[(g("_$_")[(b1 & q),(g("lb[_]:__")[b1,ws,as])]),
            (b1;b2)]),SS,CC]               ||
  [(g("_%_")[(g("_$_")[(b1 & q),(g("lb[_]:__")[(b1;b2),cs,as])]),
            (b1;b2;b3)]),SS,CC] ) . }
-- case: eq #lss((as),cs) = 0 .
open Q-PQ-2-genCheck . pr(QLOCKsys1 + INV + FACTtbu)
  eq #lss((as),cs) = 0 . red ck . close
-- case: eq (#lss((as),cs) = 0) = false .
open Q-PQ-2-genCheck . pr(QLOCKsys1 + INV + FACTtbu)
  eq pq-2-inv = inv2 . eq (#lss((as),cs) = 0) = false . red ck . close
```

The correctness of the above proof score is implied by the same discussion as the one about the module `Q-PQ-1-genCheck` except that two reductions are necessary that cover all the possible two cases in which "`#lss((as),cs) = 0`" or "`(#lss((as),cs) = 0) = false`" holds. Note that the facts declared in the module

[27] The module `Q-PQ-2-genCheck` is in the file `qlock-pq-2-ps.cafe` on the web page.

`FACTtbu` is necessary for the both cases. Note also that reifying `pq-2-inv` as `inv2` is necessary for the second case but not for the first case, i.e. `cj(inv2,S:ExState)` is not necessary for the first case.

[`Q-PQ-3-genCheck`[28]]: The reduction "`red ck .`" in the following proof score proves the third verification condition "`((cj(pq-3-inv,S) and p(S) and not(q (S))) implies (S =(*,1)=>+ SS))` for any state `S` of QLOCK with `ExState`" if it returns `($):Ind`. Note that the verification condition is on the states `S` and not on the transitions `(S,SS)`; `SS` is appears as the second argument of the built-in search predicate `_=(*,1)=>_` and be bound by the CafeOBJ system for checking the existence.

```
mod! Q-PQ-3-genCheck {
ex(Q-PQ-3 + CONSTandLITL + GENstTerm)
  op ck : -> IndTr .
  eq ck = check (
    [(g("_%_")[(g("_$_")[q,empty]),b1]),SS:ExState] ||
    [(g("_%_")[(g("_$_")[empQ,(g("lb[_]:__")[b1,(rs;ws;cs),as])]),
               (b1;b2)]),SS] ||
    [(g("_%_")[(g("_$_")[(b1 & q),(g("lb[_]:__")[b1,(rs;ws;cs),as])]),
               (b1;b2)]),SS] ) . }
open Q-PQ-3-genCheck . pr(QLOCKsys1 + INV)
  eq pq-3-inv = inv1 inv2 . red ck . close
```

Because the predicate defined via "`inv3 = qas`" is proved to be an invariant by the proof score including the module `Q-INV-3-genCheck`, any `Aid` in the queue is in `Aobs` for any reachable state. Therefor, any reachable state is an instance of the state pattern `(((A1:Aid & Q:Qu) $ ((lb[A1]: L:Label) AS:Aobs)) % A2:Aid)` of the sort `ExState`. This fact implies that the alternative script at the argument position of `check_` expanse to the term of the sort `SqSqTr` that includes the state patterns subsuming all the reachable states.[29] This, in turn, implies the correctness of the above proof score.

[`Q-PQ-4-genCheck`[30]]: The two reductions "`red ck .`" in the following proof score prove the fourth verification condition "`((cj(pq-4-inv,S) and (p(S) or q(S)) and (m(S) = 0)) implies q(S))` for any state `S` of QLOCK with `ExState`" if each of them returns `($):Ind`.

```
mod! Q-PQ-4-genCheck {ex(Q-PQ-4 + CONSTandLITL + GENstTerm)
  op ck : -> IndTr .
  eq ck = check(
     [(g("_%_")[(g("_$_")[q,empty]),b1])] ||
     [(g("_%_")[(g("_$_")[empQ,(g("lb[_]:__")[b1,(rs;ws;cs),as])]),
                (b1;b2)])] ||
     [(g("_%_")[(g("_$_")[(b1 & q),(g("lb[_]:__")[b1,(rs;ws;cs),as])]),
                (b1;b2)])] ) . }
--> case: eq #lss((as),cs) = 0 .
```

[28] The module `Q-PQ-3-genCheck` is in the file `qlock-pq-3-ps.cafe` on the web page.
[29] You can see the expanded term after the `eof` in the file `qlocik-pq-3-ps.cafe`.
[30] The module `Q-PQ-4-genCheck` is in the file `qlock-pq-4-ps.cafe` on the web page.

```
open Q-PQ-4-genCheck . pr(INV)
  eq pq-4-inv = inv2 . eq #lss((as),cs) = 0 . red ck . close
--> case: eq (#lss((as),cs) = 0) = false .
open Q-PQ-4-genCheck . pr(INV)
  eq pq-4-inv = inv2 . eq (#lss((as),cs) = 0) = false . red ck . close
```

The correctness of the above proof score is implied by the same discussion as the one about the module `Q-PQ-3-genCheck` except that two reductions are necessary that cover all the possible two cases in which "`#lss((as),cs) = 0`" or "`(#lss((as),cs) = 0) = false`" holds. Note that reifying `pq-4-inv` as `inv2` is necessary, but no need to use the facts in the module `FACTtbu`.

7 Related Works and Future Issues

7.1 Related works

There are many researches on verifications of transition systems, and we only give a brief general view and point out most related works based on Maude [16].

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, 22] 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 a deductive method with algorithmic combinatorial generations of cover sets. Moreover reduction with equations is only one deductive mechanism.

Maude [16] is a sister language of CafeOBJ and both languages share many important features. The idea underlies the transition specification (Σ, E, TR) and the transition system $((T_\Sigma)_{\mathtt{State}}/(=_E)_{\mathtt{State}}, \rightarrow_{TR}, In)$ in Section 2.3 is same as the one for the topmost rewrite theory [23, 24, 17]. 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, 7]. They are based on narrowing with unification, whereas the generate & check method is based on cover sets with ordinary matching and reduction.

7.2 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 1.

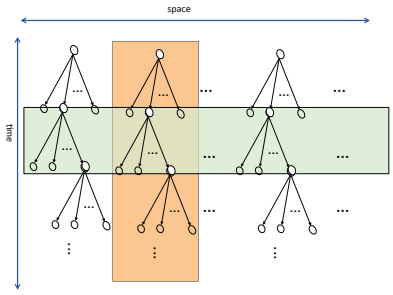

Fig. 1. Searches on Time versus Space

7.3 Future Issues

The generate & check method is more important for large and/or complex systems, for it is difficult to do case analyses manually for them. Once a state configuration is properly designed, a large number of patterns (i.e. elements of a cover set) that cover all possible cases can be generated and checked. The generic proof scores (i.e. the seven parameterized modules) in Section 4 has a potential to make the applications of the the generate & check method easy and transparent. Although, we are still in an early stage of applying the generic proof scores, the following can be expected.

- The verifications of more larger and/or complex systems become possible.
- The achieved proof scores are more well structured and transparent, and have a high potential to be good quality verification documents.

It is interesting and important future issues to investigate to what extent the above expectations will be realized.

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://cafeobj.org/ (2015)
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 (2015)
7. 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)

8. 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)
9. Futatsugi, K.: Fostering proof scores in CafeOBJ. In: Dong, J.S., Zhu, H. (eds.) ICFEM. Lecture Notes in Computer Science, vol. 6447, pp. 1–20. Springer (2010)
10. Futatsugi, K.: Generate & check method for verifying transitions systems in CafeOBJ. Lecture Notes in Computer Science, vol. 8950. Springer (accepted) (2015)
11. Futatsugi, K., Găină, 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 (2015)
16. Maude: http://maude.cs.uiuc.edu/ (2015)
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. PVS: http://pvs.csl.sri.com (2015)
23. Rocha, C., Meseguer, J.: Proving safety properties of rewrite theories. technical report. Tech. rep., University of Illinois at Urbana-Champaign (2010)
24. 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)
25. TeReSe (ed.): Term Rewriting Systems, Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press (2003)