Generic Proof Scores for Generate & Check Method in CafeOBJ

Kokichi FUTATSUGI
Japan Advanced Institute of Science and Technology

26 February 2015
JAIST, Nomi, Ishikawa, Japan

You can find these slides, a paper, and CafeOBJ codes for the talk on the following web page.

`http://www.jaist.ac.jp/~kokichi/talk/150226jaistVsemi/`

Specification Verification

- Constructing specifications and verifying them in the upstream of system/software development are still one of the most important challenges in system/software 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.

Verification with Proof Scores (1)

- A system and the system's properties are specified in an executable algebraic specification language (CafeOBJ in our case).
- Proof scores are described also in the same executable 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 (i.e. rewriting from left to wright) 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.

Verification with Proof Scores (2)

- The concept of proof supported by proof scores is similar to that of Larch Prover or Maude ITP. Larch's specification language is, however, not executable.
- Proof scripts written in tactic languages provided by theorem provers such as Coq and Isabelle/HOL have similar nature as proof scores.
- Proof scores are written, however, 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.

Generate & Check Method (1)

- For a sort Srt and a predicate p on Srt we get $((p(X{:}Srt) \to_E^* \texttt{true})\ \texttt{implies}\ (\forall t \in (T_\Sigma)_{Srt})(p(t) =_E \texttt{true}))$ and $(p(X{:}Srt) \to_E^* \texttt{true})$ is a sufficient condition to prove $(\forall t)p(t)$.
- However, usually p is not simple enough to obtain $(p(X{:}Srt) \to_E^* \texttt{true})$ 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** $(p(t_i) \to_E^* \texttt{true})$ for each $i \in \{1, \cdots, m\}$.
- **Induction** is another technique for proving $(p(X{:}Srt) \to_E^* \texttt{true})$ for a constrained sort Srt.

Generate & Check Method (2)

- The generation & checking can be a theorem proving method for transition systems based on
  (1) generation of finite state patters that cover all possible infinite states, and
  (2) checking the validities of verification conditions for each of the finite state patterns.
- The state space of a transition system is formalized as a quotient set (i.e. a set of equivalence classes) of terms of a topmost sort *State*, and the transitions are specified with conditional rewrite rules over the quotient set.

Generate & Check Method (3)

A property to be verified is either

- an invariant (i.e. a state predicate that is valid for all reachable states), or
- a (p leads-to q) property for two state predicates p and q ((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).

Generic Proof Scores for the Generate & Check Method have been developed

- 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.

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).

- 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.
- 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**.
- 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 `Bool`) is defined to be an **invariant** (or an invariant property) if $(p(s^r) = \texttt{true})$ for any reachable state s^r.

- Let $\Sigma = (S, \leq, F)$ be a regular order-sorted signature with a set of sorts S, and let $X = \{X_s\}_{s \in S}$ be an S-sorted set of variables.
- Let $T_\Sigma(X)$ be S-sorted set of $\Sigma(X)$-terms, let $T_\Sigma(X)_s$ be a set of $\Sigma(X)$-terms of sort s, let E be a set of $\Sigma(X)$-equations, and let (Σ, E) be an equational specification with a unique sort `State`.
- 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 the morphism from $T_\Sigma(X)$ to $T_\Sigma(Y)$, and $t\,\theta$ is the term obtained by substituting $x \in X$ in t with $x\,\theta$.

- 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.
- 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.

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

- 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 $In \subseteq (T_\Sigma/{=_E})_{\texttt{State}}$. In is assumed to be defined via a state predicate *init* that is defined with E, i.e. $(s \in In)$ `iff` $(init(s) =_E \texttt{true})$.
- Then a transition specification (Σ, E, TR) defines a transition system $((T_\Sigma/{=_E})_{\texttt{State}}, \rightarrow_{TR}, In)$.

- Given a transition system $TS = (St, Tr, In)$, and let $p_1, p_2, \cdots, p_n$ $(n \in \{1, 2, \cdots\})$ be state predicates of TS, and $inv(s) \stackrel{\text{def}}{=} (p_1(s) \texttt{ and } p_2(s) \texttt{ and } \cdots \texttt{ and } p_n(s))$ for $s \in St$.
- The following three conditions are sufficient for a state predicate p^t to be an invariant.

  (1) $(\forall s \in St)(inv(s) \texttt{ implies } p^t(s))$
  (2) $(\forall s \in St)(init(s) \texttt{ implies } inv(s))$
  (3) $(\forall (s, s') \in Tr)(inv(s) \texttt{ implies } inv(s'))$

- 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.

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).

- 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).
- Let $TS = (St, Tr, In)$ be a transition system, and let p, q be predicates with arity $(St, Data)$ of TS, where $Data$ is a data sort needed to specify p, q. A transition system is defined to have the (p **leads-to** q) **property** if and only if the system will get to 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 a liveness property, and is adopted from the UNITY logic.

Based on an original transition system $TS = (St, Tr, In)$, let

$$\widehat{St} \stackrel{\text{def}}{=} St \times Data,$$
$$(((s, d), (s', d)) \in \widehat{Tr}) \stackrel{\text{def}}{=} ((s, s') \in Tr),$$
$$\widehat{In} \stackrel{\text{def}}{=} In \times Data,$$
$$\widehat{TS} \stackrel{\text{def}}{=} (\widehat{St}, \widehat{Tr}, \widehat{In}).$$

Let *inv*1, *inv*2, *inv*3 *inv*4 be invariants of $\widehat{TS}$ and
let m be a function from $\widehat{St}$ to `Nat` (the set of natural numbers),
then the 4 conditions in the next slide are sufficient for
the (p leads-to q) property to be valid for $\widehat{TS}$. Here

$$\widehat{s} \stackrel{\text{def}}{=} (s, d) \text{ for any } d \in Data,$$
$$p(\widehat{s}) \stackrel{\text{def}}{=} p(s, d) \text{ and } q(\widehat{s}) \stackrel{\text{def}}{=} q(s, d).$$

The four Sufficient Verification Conditions for (p leads-to q) Properties

(1) $(\forall(\widehat{s},\widehat{s'}) \in \widehat{Tr})$
$((inv1(\widehat{s}) \texttt{ and } p(\widehat{s}) \texttt{ and } (\texttt{not } q(\widehat{s}))) \texttt{ implies } (p(\widehat{s'}) \texttt{ or } q(\widehat{s'})))$

(2) $(\forall(\widehat{s},s') \in \widehat{Tr})$
$((inv2(\widehat{s}) \texttt{ and } p(\widehat{s}) \texttt{ and } (\texttt{not } q(\widehat{s}))) \texttt{ implies } (m(\widehat{s}) > m(\widehat{s'})))$

(3) $(\forall\widehat{s} \in \widehat{St})$
$((inv3(\widehat{s}) \texttt{ and } p(\widehat{s}) \texttt{ and } (\texttt{not } q(\widehat{s}))) \texttt{ implies }$
$(\exists\widehat{s'} \in \widehat{St})((\widehat{s},\widehat{s'}) \in \widehat{Tr}))$

(4) $(\forall\widehat{s} \in \widehat{St})$
$((inv4(\widehat{s}) \texttt{ and } (p(\widehat{s}) \texttt{ or } q(\widehat{s})) \texttt{ and } (m(\widehat{s}) = 0)) \texttt{ implies } q(\widehat{s}))$

[Subsume] 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 (i.e. terms without variables) $G \subseteq T_\Sigma$ iff for any $t' \in G$ there exits $t \in C$ such that t' is an instance of t.

[Generate&Check-S] Let $(T_\Sigma/{=_E})_{\texttt{State}}, \rightarrow_{TR}, In)$ be a transition system defined by a transition specification (Σ, E, TR). 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 $s \in C$. □

Let `q` be a predicate with arity "`State State`" for stating some relation of the current state and the next state, like ($inv(s)$ `implies` $inv(s')$). Let the function `valid-q` be defined using the CafeOBJ 's built-in search predicate

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

as follows.

```
-- 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 $s \in T_{\Sigma}(Y)_{\mathtt{State}}$, 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 $tr_j = (\forall X)(l_j \rightarrow r_j \text{ if } c_j)$ in Tr and a substitution $\theta \in T_{\Sigma}(Y)^{X}$ such that $\mathtt{s} = l_j\,\theta$.
2. For each found (tr_j, θ), let $(\mathtt{SS} = r_j\,\theta)$ and $(\mathtt{CC} = c_j\,\theta)$ and print out `(ifm s SS CC q(s,SS))` and tr_j if $(\mathtt{not((CC\ then\ q(s,SS))\ ==\ true)} \twoheadrightarrow^{*}_{E} \mathtt{true})$.
3. Returns `false` if any print out exits, and returns `true` otherwise.

[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'.

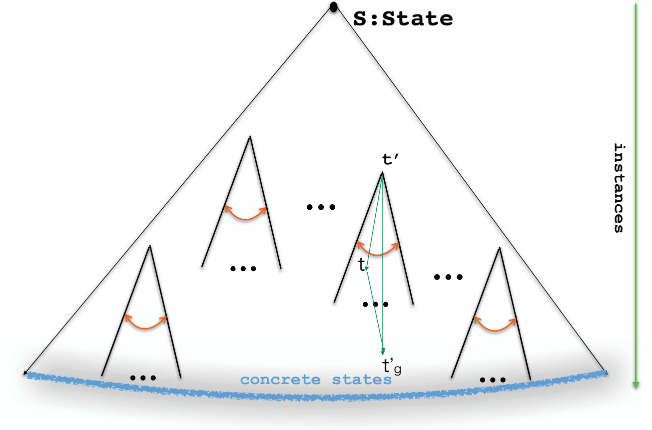
[Generate&Check-T1] Let $((T_\Sigma/{=_E})_{\mathtt{State}}, \rightarrow_{TR}, In)$ be a transition system, and let $C' \subseteq T_\Sigma(X)$ be the set of all the left-hand sides of the transition rules in TR. Then doing the following **Generate** and **Check** are sufficient for verifying

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

for a predicate "$\mathtt{pred\ q_{tr}\ :\ \ State\ State}$".

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

Covering



[Generate&Check-T2] Let $TR = \{tr_1, \cdots, tr_m\}$ be a set of transition rules, and let $tr_i = (\forall X)(l_i \to r_i \texttt{ if } c_i)$ 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

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

for a predicate "$\mathtt{pred\ q_{tr}\ :\ \ State\ State}$".

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

The conditions (1) and (2) for invariant properties 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.

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

Note that, if $inv \stackrel{\text{def}}{=} (p_1 \ \texttt{and} \cdots \texttt{and} \ p_n)$ and $p^t = (p_{i_1} \ \texttt{and} \cdots \texttt{and} \ p_{i_m})$ for $\{i_1, \cdots, i_m\} \subseteq \{1, \cdots, n\}$, then condition (1) is directly obtained.

The condition (3) for invariant properties can be verified by using Generate&Check-T1 or T2 with $\mathtt{q}_{\mathtt{tr\text{-}3}}(s, s')$ defined as follows.

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

The conditions (1) and (2) for the (p leads-to q) properties can be verified by using Generate &Check-T1 or T2 in Section 23 with $\mathtt{q}_{\mathtt{tr-1}}(\widehat{s}, \widehat{s'})$ and $\mathtt{q}_{\mathtt{tr-2}}(\widehat{s}, \widehat{s'})$ defined as follows respectively.

$$(1)\quad \mathtt{q}_{\mathtt{tr-1}}(\widehat{s}, \widehat{s'}) = ((inv1(\widehat{s}) \texttt{ and } p(\widehat{s}) \texttt{ and } (\texttt{not } q(\widehat{s}))) \texttt{ implies } (p(\widehat{s'}) \texttt{ or } q(\widehat{s'})))$$

$$(2)\quad \mathtt{q}_{\mathtt{tr-2}}(\widehat{s}, \widehat{s'}) = ((inv2(\widehat{s}) \texttt{ and } p(\widehat{s}) \texttt{ and } (\texttt{not } q(\widehat{s}))) \texttt{ implies } (m(\widehat{s}) > m(\widehat{s'})))$$

The conditions (3) and (4) for (p leads-to q) properties can be verified by using Generate &Check-S in Section 20 with $\mathtt{p_{st\text{-}3}}(\widehat{s})$ and $\mathtt{p_{st\text{-}4}}(\widehat{s})$ defined as follows respectively.

$$(3)\ \ \mathtt{p_{st\text{-}3}}(\widehat{s}) = ((inv3(\widehat{s})\ \mathtt{and}\ p(\widehat{s})\ \mathtt{and}\ (\mathtt{not}\ q(\widehat{s})))\ \mathtt{implies}\ (\widehat{s}\ \mathtt{=(*,1)=+\ SS:State}))$$

$$(4)\ \ \mathtt{p_{st\text{-}4}}(\widehat{s}) = ((inv4(\widehat{s})\ \mathtt{and}\ (p(\widehat{s})\ \mathtt{or}\ q(\widehat{s}))\ \mathtt{and}\ (m(\widehat{s}) = 0))\ \mathtt{implies}\ q(\widehat{s}))$$

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

7 Parameterized Modules for 7 Verification Conditions

- The seven parameterized CafeOBJ modules codify the seven verification conditions of the generate & check method. The seven verification conditions are the three verification conditions for invariant properties and the four verification conditions 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, basic parts of proof scores are obtained. As a result, proof score developers can concentrate on case analyses and lemma discoveries that require human insight.

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

- The function `check_` 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`.

```
-- SqSq enclosures SqSqEn and their trees SqSqTr
[SqSqEn < SqSqTr]
op [_] : SqSq -> SqSqEn .
op _||_ : SqSqTr SqSqTr -> SqSqTr {assoc strat: (1 0)}
```

- 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 `_||_`with the strategy `(1 0)`. An `SqSqEn` is an `SqSq` enclosed with `[` and `]`.
- A term composed of an associative binary operator inductively is called a tree.

```
[Val < ValSq] op _,_ : ValSq ValSq -> ValSq {assoc}
[Val < VlSq] op _;_ : VlSq VlSq -> VlSq {assoc strat: (1 0)}
[ValSq VlSq < SqSq] op empSS : -> SqSq .
op _,_ : SqSq SqSq -> SqSq {assoc id: empSS}
```

- 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 `VlSq` is (1) a `Val` or (2) a sequence of `VlSq`s composed of the associative binary operator `_;_`with the strategy `(1 0)`.
- A `ValSq` is (1) a `Val` or (2) a sequence of `Val`s composed of the associative binary operator `_,_`.
- The operator `_,_` is overloaded (i.e. denotes two different operations). A term composed of an associative binary operator inductively is called a sequence.

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`.

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) ]
```

```
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) .
```

- To make the alternative expansion with _;_ more versatile, the functions `t__` and `g__` are introduced. `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`.

Let the following equations for `t__` be given.

```
[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 a term of sort `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`.

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`.

```
[Info] op i__ : Bool Info -> Ind .
eq (i true II:Info) | IT:IndTr = IT .
op ii_ : ValSq -> Info .
pred v_ : ValSq .
op mi_ : ValSq -> Ind .
eq mi(VSQ:ValSq) = (i v(VSQ) ii(VSQ)) .
```

For defining conjunctions of predicates flexibly, the following parameterized module PREDcj is prepared.

```
mod! PREDcj (X :: TRIV) {
[Pname < PnameSeq]
op _ _ : PnameSeq PnameSeq -> PnameSeq {constr assoc}
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 verification conditions (1),(2),(3) for invariants the verification conditions (1),(2),(3),(4) for (p leads-to q) properties.

The following two parameterized modules `INV-1v` and `INV-2v` codify the verification conditions (1) and (2) for invariants directory. The `PnameSeq`s `p-iinv`, `p^t`, and `p-init` are supposed to be reified after the parameter modules are substituted with actual specification modules.

```
mod* STEpcj {[Ste] [Pname < PnameSeq]
             pred cj : PnameSeq Ste .}
mod! INV-1v (ST :: STEpcj) {ex(GENcases)
 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) . }
```

The following parameterized module `VALIDq` directly specifies `valid-q`. `inc(RWL)` declares the importation of the built-in module `RWL` that is necessary for using the built-in search predicate.

```
mod* STE {[Ste]}
mod! VALIDq (X :: STE) {inc(RWL)
 pred q : Ste Ste .
 [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`.
- 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)) . }
```

- For specifying the four verification conditions for (p leads-to q) properties, the states are needed to extend with data. The parameterized module `EX-STATE` in the next slide specifies the state extension.
- 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 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.

```
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] op _%_ : Ste Data -> ExState {constr}
 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) .
 ops p q : ExState -> Bool . op m : ExState -> Nat.PNAT .
     eq p(S:Ste % D:Data) = p(S,D) .
     eq q(S:Ste % D:Data) = q(S,D) .
     eq m(S:Ste % D:Data) = m(S,D) .
 [Ste Data ExState < Val]
     eq t("_%_")(S:Ste,D:Data) = (S % D) . }
```

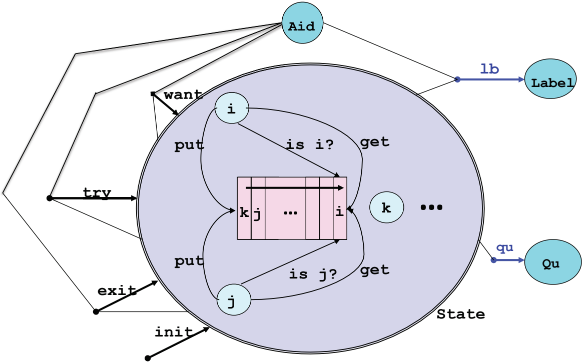
The following parameterized module `PCJ-EX-STATE` makes the `cj` available on `ExState` and relate that to the `cj` on `Ste`.

```
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) . }
```

- The four parameterized modules for the four verification conditions for (p leads-to q) properties are specified in the following two slids. These are direct translation from the four verification conditions.
- 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 .
  eq q(S:Ste,SS:Ste) =
     (cj(pq-2-inv,S) and p(S) and not(q(S)))
                     implies (m(S) > m(SS)) . }
```

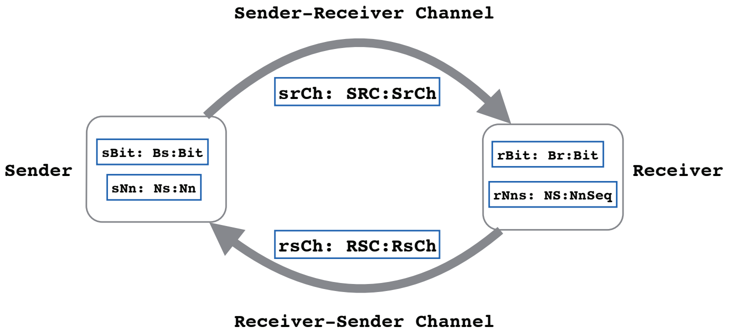
```
mod! PQ-3v (SQ :: STPQpcj) {inc(RWL) ex(GENcases)
  op pq-3-inv : -> PnmSeq .  [Ste < Val]
  eq v(S:Ste,SS:Ste) =
     (cj(pq-3-inv,S) and p(S) and not(q(S)))
                         implies (S =(*,1)=>+ SS) . }
mod! PQ-4v (SQ :: STPQpcj) {pr(GENcases)
  op pq-4-inv : -> PnmSeq . [Ste < Val]
  eq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



Global view of QLOCK as an Observational Transition System

```
-- 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
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)}
```

Alternating Bit Protocol



```
- State patterns/configurations are represented as
  follows.
   [(sBit: Bs:Bit)(sNn: Ns:Nn)    (srCh: SRC:SrCh)
    (rBit: Br:Bit)(rNns: NS:NnSeq)(rsCh: RSC:RsCh)]

- In the following CafeOBJ specifications,
  data (i.e. Bnp and Bit) are getting through
  SrCh (sender-reciever channel) and
  RsCh (reciever-sender channel) from left to right.
```

```
-- Sender is putting a bit-number pair into SrCh
mod! SS {pr(CONFIG) trans[ss]:
   [(sBit: B:Bit)(sNn: N:Nn)(srCh: SRC:SrCh) S:Config]
=> [(sBit: B)(sNn: N)(srCh: (bn(B,N) SRC)) S] . }
-- Sender is receiving a bit (an ack) from RsCh
mod! SR {pr(CONFIG) trans[sr]:
  [(sBit: Bs:Bit)(sNn: N:Nn) S:Config
   (rsCh: (RSC:RsCh B:Bit))]
=> if (Bs = B)
     then [(sBit: Bs)(sNn: N) S (rsCh: RSC)]
     else [(sBit: not(Bs))(sNn: (s N)) S
           (rsCh: RSC)] fi . }
```

```
-- data drops at any point of srCh
mod! SRdr {pr(CONFIG) trans[srdr]:
   [S1:Config (srCh: (SRC1:SrCh BNP:Bnp SRC2:SrCh))
    S2:Config]
=> [S1 (srCh: (SRC1 SRC2)) S2] . }
-- data duplicates at any point of srCh
mod! SRdu {pr(CONFIG) trans[srdu]:
   [S1:Config (srCh: (SRC1:SrCh BNP:Bnp SRC2:SrCh))
    S2:Config]
=> [S1 (srCh: (SRC1 BNP BNP SRC2)) S2] . }
```

```
-- Receiver is receiving a bit-number pair from srCh
mod! RR {pr(CONFIG) trans[rr]:
[S1:Config (srCh: (SRC:SrCh bn(B:Bit,N:Nn)))
 (rBit: Br:Bit)(rNns: NS:NnSeq) S2:Config]
=> if (B = Br)
   then [S1 (srCh: SRC)(rBit: not(Br))(rNns: (N NS))
         S2]
   else [S1 (srCh: SRC)(rBit: Br)(rNns: NS) S2] fi . }
-- Receiver is sending a number to RsCh
mod! RS {pr(CONFIG) trans[rs]:
   [S:Config (rBit: B:Bit)(rsCh: RSC:RsCh)]
=> [S (rBit: B)(rsCh: (B RSC))] . }
```

```
-- data drops at any point of rsCh
mod! RSdr {pr(CONFIG) trans[rsdr]:
    [S1:Config (rsCh: (RSC1:RsCh B:Bit RSC2:RsCh))
     S2:Config]
=> [S1 (rsCh: (RSC1 RSC2)) S2] . }
-- data duplicates at any point of rsCh
mod! RSdu {pr(CONFIG) trans[rsdu]:
    [S1:Config (rsCh: (RSC1:RsCh B:Bit RSC2:RsCh))
     S2:Config]
=> [S1 (rsCh: (RSC1 B B RSC2)) S2] . }
```

- <SR1>, <RS1>, <SR2>, or <RS2> represents some SrCh or RsCh that satisfies the following equations.

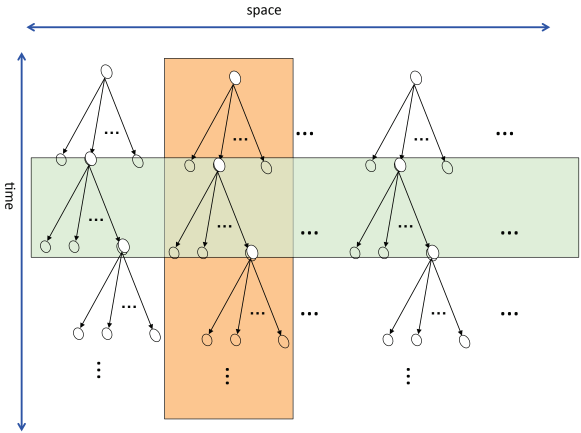
```
eq Og(dn(B,N), <SR1>) = true .
eq Og(B, <RS1>) = true .
eq Og(dn(~B,(s N)), <SR2>) = true .
eq Og(~B, <RS2>) = true .
```

- (mk(N) = (N NS)) and (~B = not(B)).

```
[(sBit: B) (sNn: N)       (srCh: <SR1>)
    {SS,SRdr,SRdu,RS,RSdr,RSdu,SR}
 (rBit: B) (rNns: NS)     (rsCh: <RS1>)]
 {RR}=>
[(sBit: B) (sNn: N)       (srCh: <SR1>)
    {SS,SRdr,SRdu,RR,RSdr,RSdu,SR}
 (rBit: ~B)(rNns: (N NS))(rsCh: <RS1>)]
 {RS}=> <={RSdr}
[(sBit: B) (sNn: N)       (srCh: <SR1>)
    {SS,SRdr,SRdu,RR,RSdr,RSdu,SR}
 (rBit: ~B)(rNns: (N NS))(rsCh: <RS2><RS1>)]
```

```
[(sBit: B) (sNn: N)       (srCh: <SR1>)
    {SS,SRdr,SRdu,RR,RSdr,RSdu,SR}
 (rBit: ~B)(rNns: (N NS))(rsCh: <RS2><RS1>)]
 {SR,RSdr}=>
[(sBit: B) (sNn: N)       (srCh: <SR1>)
    {SS,SRdr,SRdu,RR,RS,RSdr,RSdu}
 (rBit: ~B)(rNns: (N NS))(rsCh: <RS2>)]
 {SR}=>
[(sBit: ~B)(sNn: (s N))  (srCh: <SR1>)
    {SRdr,SRdu,RR,RS,RSdr,RSdu,SR}
 (rBit: ~B)(rNns: (N NS))(rsCh: <RS2>)]
```

```
[(sBit: ~B)(sNn: (s N))  (srCh: <SR1>)
    {SRdr,SRdu,RR,RS,RSdr,RSdu,SR}
 (rBit: ~B)(rNns: (N NS))(rsCh: <RS2>)]
 {SS}=> <={SRdr}
[(sBit: ~B)(sNn: (s N))  (srCh: <SR2><SR1>)
    {SS,SRdr,SRdu,RR,RS,RSdr,RSdu}
 (rBit: ~B)(rNns: (N NS))(rsCh: <RS2>)]
 {RR,SRdr}=>
[(sBit: ~B)(sNn: (s N))  (srCh: <SR2>)
    {SS,SRdr,SRdu,RS,RSdr,RSdu,SR}
 (rBit: ~B)(rNns: (N NS))(rsCh: <RS2>)]
```



Searches on Time versus Space

- There are recent attempts to extend the model checking with Maude for verifying infinite state transition systems. They are based on narrowing with unification, whereas the generate & check method is based on cover sets with ordinary matching and reduction.
- 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.
- Module expressions of CafeOBJ is powerful, and are expected to be effective for constructing large specifications/ proof-scores with systematic structures.