**1.**

There are three kinds of *variables*: 1) simple identifier, denoted by a string and usually a character; 2) element of an array, denoted by a string followed by a natural inside a square bracket. E.g., $arr[i]$ indicates the $i$th element of the array $arr$; 3) filed of a record, denoted by a string followed by a dot and then another string. E.g., $rcd.f$ indicates the filed $f$ of the record $rcd$. Each variable is associated with its *type*. There are three types in our model, including enumeration, natural, and boolean.

*Experssions* can be simple or compound. A simple expression is either a variable or a constant, while a compound expression is constructed with the form $f?e_1 : e_2$, where $e_1$ and $e_2$ are expressions. A $formula$ can be an atomic formula or a compound formula. An atomic formula can be a boolean variable or boolean constant, or in the equivalence form $e_1 = e_2$, where $e_1$ and $e_2$ are two expressions. A $formula$ can also be constructed from formulas using the logic connectives, including negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$).

An *assignment* is a mapping from a variable to an expression, and is denoted with the assigning operation symbol ":=". If an assignment maps a variable to a (constant) value, then we say it is a *value-assignment*. A *statement* $\alpha$ is a set of assignments which are executed in parallel, e.g., $\{x_1 := e_1; x_2 := e_2; ...; x_k := e_k\}$. We use $\alpha|_x$ to denote the expression assigned to $x$ under the statement $\alpha$. For example, let $\alpha$ be $\{arr[1] := C; x := false\}$, then $\alpha|_x$ returns $false$. A *state* is an instantaneous snapshot of its behavior given by a set of value-assignments.

For every expression $e$ and formula $f$, we denote the value of $e$ (or $f$) under the state $s$ as $\mathbb{A}[e, s]$ (or $\mathbb{B}[f, s]$) For an expression $e$ and a formula $f$, we write $s, e \mapsto c$ and $s \models f$ to mean $\mathbb{A}[e, s] = c$ and $\mathbb{B}[f, s] = true$. Formal semantics of expressions and formulas are given as follows, which hold for any state $s \in S$.

| Semantics |
|---|
| $\mathbb{A}[v, s] \equiv s(v)$, where $v$ is a variable |
| $\mathbb{A}[c, s] \equiv c$, where $c$ is a constant |
| $\mathbb{A}[f?e_1 : e_2, s] \equiv$if ($\mathbb{B}[f, s]$) then $\mathbb{A}[e_1, s]$ else $\mathbb{A}[e_2, s]$ |
| $\mathbb{B}[e_1 = e_2, s] \equiv \mathbb{A}[e_1, s] = \mathbb{A}[e_2, s]$ |
| $\mathbb{B}[\neg f, s] \equiv \mathbb{B}[f, s] = false$ |
| $\mathbb{B}[f_1, \wedge f_2, s] \equiv \mathbb{B}[f_1, s]$ and $\mathbb{B}[f_1, s]$ |
| $\mathbb{B}[f_1, \vee f_2, s] \equiv \mathbb{B}[f_1, s]$ or $\mathbb{B}[f_1, s]$ |

For an expression $e$ and a statement $\alpha = \{x_1 := e_1; x_2 := e_2; ...; x_k := e_k\}$, we use $Var(\alpha)$ to denote the variables to be assigned $\{x_1, x_2, ...x_k\}$; and use $e^\alpha$ to denote the expression transformed from $e$ by substituting each $x_i$ with $e_i$ simultaneously. Similarly, for a formula $f$ and a statement $\alpha = \{x_1 := e_1; x_2 := e_2; ...; x_k := e_k\}$, we use $f^\alpha$ to denote the formula transformed from $f$ by substituting each $x_i$ with $e_i$. Moreover, $f^\alpha$ can be regarded as the weakest precondition of formula $f$ w.r.t. statement $\alpha$, and we denote $preCond(f, \alpha) = f^\alpha$. Noting that a state transition is caused by an execution of the statement (given that the guard is satisfied).

A *rule* $r$ is a pair $< g, \alpha >$, where $g$ is a formula and is called the *guard* of rule $r$, and $\alpha$ is a statement and is called the *action* of rule $r$. For convenience, we denote a rule with the guard $g$ and the statement $\alpha$ as $g \rhd \alpha$, and $act(g \rhd \alpha) = \alpha$ and $guard(g \rhd \alpha) = g$. If the guard $g$ is satisfied at state $s$, then $\alpha$ can be executed and a new state $s'$ is derived. We call the rule $g \rhd \alpha$ is triggered at $s$, and the protocol transits from $s$ into $s'$. Formally, for a rule $r = g \rhd \alpha$, we define $s \xrightarrow{r} s'$ iff 1) $s \models guard(r)$ and 2) $\forall x \in Var(\alpha).s'(x) = \mathbb{A}[e, s]$, where $e$ is the assignment to $x$ under $\alpha$.

A *protocol* $\mathcal{P}$ is a pair $(I, R)$, where $I$ is a set of $formulas$ and is called the initializing formula set, and $R$ is a set of rules. As usual, the reachable state set of protocol $\mathcal{P} = (I, R)$, denoted as $\mathcal{R}(\mathcal{P})$, can be defined inductively: (1) a state $s$ is in $\mathcal{R}(\mathcal{P})$ if $s \models f$ for some formula $f \in I$; (2) a state $s$ is in $\mathcal{R}(\mathcal{P})$ if there exists a state $s_0$ and a rule $r \in R$ such that $s_0 \in \mathcal{R}(\mathcal{P})$ and $s_0 \xrightarrow{r} s$.

Now we use a simple example to illustrate the above definitions by a simple mutual exclusion protocol with $N$ nodes. Let $\mathsf{I} \equiv$ `enum` "*control*" "$I$",

## 2. The Searching Algorithm

In this section, we present an algorithm called `InvFinder`, which finds all necessary ground invariants from a protocol instance. As mentioned before, initially there is only one invariant in the invariant set, which is a *mutualInv* formula. The algorithm `InvFinder` works iteratively in a semi-proving and semi-searching fashion to create invariant, until no new invariant is created. In each iteration, it calls a function named `findInvFromRules`, trying to prove some consistent relation between an invariant and a rule, and automatically generates a new auxiliary invariant if there is no such an invariant in the invariant set, and records the corresponding causal relation information between the current rule and invariant.

The core of `InvFinder` is the `findInvFromRules` function, and this section focuses on this algorithm. The `findInvFromRules` algorithm needs to call two oracles. The first one, denoted by `chk`, checks whether a ground formula is an invariant in a given small reference model of the protocol. Such an oracle can be implemented by translating the formula into a formula in SMV, and calling SMV to check whether it is an invariant. The second oracle, denoted by `tautChk`, checks whether a formula is a tautology. Such a tautology checker is implemented by translating the formula into a form in the SMT (abbreviation for SAT Modulo Theories) format, and then calls an SMT solver such as Z3 to check it.

Besides the two oracles which are passed as parameters, there are other parameters in the algorithm `findInvFromRules`, including a rule instance *rule*, an invariant *inv*, two sets of invariants *invs* and *newInvs*, and a set of causal relations *casRel*. The algorithm `InvFinder` searches for new invariants and constructs the causal relation between the rule instance *rule* and the invariant *inv*. The sets *invs* and *newInvs* store ..., and the set *casRel* stores causal relations constructed up to now. The algorithm `findInvFromRules` returns new invariants and causal relations.

After computing the pre-condition $inv'$ w.r.t. the input invariant and the statement of the input rule, the algorithm performs case analysis on $inv'$ and takes further operations according to the case it faces with.

(1) If `inv=inv'`, which means that statement $S$ does not change $inv$, then no new invariant is created, and a new causal relation item marked with tag $\texttt{invRule}_2$ is recorded between $rule$ and $inv$.

For instance, let `rule=crit 3`, `inv=mutualInv 1 2`, then `inv'=preCond(inv,S)=inv`. In this case, only a new relation item $\texttt{(crit 3, inv, invRule}_2\texttt{,\_)}$ will be added.

(2) If $tautChk$ verifies that $g \rightarrow inv'$ is a tautology, then no new invariant is created, and a new causal relation item marked with tag $\texttt{invRule}_1$ is recorded between $rule$ and $inv$. For instance, let `rule=crit 2`, $\texttt{inv=invOnX}_1\texttt{ 1}$, then $\texttt{inv'=preCond(inv,S)=}\neg\texttt{(false=true} \wedge n[1] = C\texttt{)}$. Obviously, $g \rightarrow inv'$ always holds because $inv'$ is always evaluated true. In this case, a new relation item $\texttt{(crit 2, inv, invRule}_1\texttt{,\_)}$ will be added.

(3) If neither of the above two cases holds, then a new auxiliary invariant $newInv$ will be constructed, making the causal relation $\texttt{invRule}_3$ holds.

The construction of the auxiliary invariant is introduced better after giving some definitions. A formula $f$ can be composed into a set of sub-formulas $f_i$, denoted as $decompose(f)$, such that each $f_i$ is not of a conjunction form and $f$ is semantically equivalent to $f_1 \wedge f_2 \wedge ... \wedge f_N$. For a formula $f$, we use $subformulasets(f)$ to denote the power set of $decompose(f)$, which contains all subsets of $decompose(f)$.

A proper formula is chosen from the candidate set $subformulasets(dualNeg(inv') \wedge g)$ to construct a new invariant $newInv$. This is accomplished by the `choose` function, which calls the oracle `chk` to verify whether a formula is an invariant in the given reference model. After $newInv$ is chosen, the function $isNew$ checks whether this invariant is new w.r.t. $newInvs$ or $invs$. If this is the case, the invariant will be normalized, and then be added into $newInvs$, and the new causal relation item marked with tag $\texttt{invRule}_3$ will be added into the causal relations. Here, the meaning of the word "new" is modulo to the symmetry relation. For instance, mutualInv 1 2 is equivalent to mutualInv 2 1 in a symmetry view.

**Algorithm 1:** Searching Algorithm: $findInvFromRule$

**Input**: $chk$, $tautChk$, $rule$, $inv$, $invs$, $newInvs$, $casRel$<br>
**Output**: A formula set $F$, a rule set $R$<br>
$g \leftarrow$the guard of rule, $S \leftarrow$the action of rule;<br>
$inv' \leftarrow preCond(inv, S)$;<br>
**if** $inv = inv'$ **then**<br>
    $relItem \leftarrow (rule, inv, invRule_2, -)$;<br>
    **return** $(newInvs, relItem : casRel)$;<br>
**else if** $tautChk(g \rightarrow inv') = true$ **then**<br>
    $relItem \leftarrow (rule, inv, invRule_1, -)$;<br>
    **return** $(newInvs, relItem : casRel)$;<br>
**else**<br>
    $candidates \leftarrow subformulasets(dualNeg(inv') \wedge g)$;<br>
    $newInv \leftarrow choose(chk, candidates)$;<br>
    $relItem \leftarrow (rule, inv, invRule_3, newInv)$;<br>
    **if** $isNew(newInv, newInvs \cup invs)$ **then**<br>
        normalize $newInv$ and insert it into the head of $newInvs$;<br>
        **return** $(newInvs, relItem : casRel)$;<br>
    **else**<br>
        **return** $(newInvs, relItem : casRel)$;