# Appendix

## 1 Formal Semantics of Expressions and Formulas

Formal semantics of expressions and formulas are given in HOL (higer-order logics) as Table 1 shows. [1]

| 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 $\mathbb{A}[e_2, s]$ |
| $\mathbb{B}[e_1 \doteq e_2, s] \equiv \mathbb{A}[e_1, s] = \mathbb{A}[e_2, s]$ |
| $\mathbb{B}[!f, s] \equiv \neg\mathbb{B}[f, s]$ |
| $\mathbb{B}[f_1 \barwedge f_2, s] \equiv \mathbb{B}[f_1, s] \wedge \mathbb{B}[f_1, s]$ |
| $\mathbb{B}[f_1 \veebar f_2, s] \equiv \mathbb{B}[f_1, s] \vee \mathbb{B}[f_2, s]$ |
| $\mathbb{B}[f_1 \dashrightarrow f_2, s] \equiv \mathbb{B}[f_1, s]$ implies $\mathbb{B}[f_2, s]$ |

## 2 Computing a Quotient of $\mathsf{perms}_m^n$

Algorithm 1 computes a quotient of $\mathsf{perms}_m^n$. Firstly it set $S_0 = \mathsf{perms}_m^n$, then we fetch the head element of $S_0$ into $L$, and find whether there is an element $L'$ in $S$ s.t. $L \sim_m^n L'$. If yes, then $L$ will be discarded, else $L$ is inserted into $S$. This procedure is repeated until $S$ is empty.

In order to understand this algorithm, we need understand the following:

- a $n$-permutation of $m$ is ordered arrangement of a $n$–element subset of an $m$-element set $I = \{i.0 < i \leq m\}$. We use a list with size $n$ to stand for a $n$-permutation of $m$, whose elements are mutually different from each other and taken from $I$. For instance, $[1, 2]$ is a 2-permuatation of 3. $\mathsf{perms}_m^n$ is the set of all $n$-permutations of $m$. There have been a lot of algorithms to compute $\mathsf{perms}_m^n$, so we only list examples: $\mathsf{perms}_m^0 = []$, $\mathsf{perms}_3^2 = [[1, 2], [1, 3], [2, 1], [2, 3], [3, 1], [3, 2]]$. Number of elements in $\mathsf{perms}_m^n = \frac{m!}{(m-n)!}$.
- According to the definition of $simeq_m^n$ in the main paper, we list some examples to illustrate the the above algorithm.

[1] The logic to specify parameterized system is a special logic, which can be embedded in HOL supported by Isabelle. Therefore HOL can be seen as the main meta-logic to specify our work.

**Algorithm 1:** Computing quotient of $\mathsf{perms}_m^n$: *cmpSemiperm*

**Input**: $m$, $n$
**Output**: A permutation set $S$
$S_0 \leftarrow \mathsf{perms}_m^n$;
$S \leftarrow \emptyset$;
**while** $S_0 \neq \emptyset$ **do**
    $L \leftarrow \mathsf{hd}(S_0)$;
    $S_0 \leftarrow \mathsf{tl}(S_0)$;
    **if** $\mathsf{find}(\simeq_m^n (L), S) = NONE$ **then**
        $S \leftarrow S@[L]$;
**return** $S$;

- When $m = 3$ and $n = 2$, then the output should be the same as $\mathsf{perms}_3^2$.
- When $m = 3$ and $n = 1$, then the output should be the same as $\mathsf{perms}_3^1$.
- When $m = 4$ and $n = 2$, then the output should be $[[1, 2], [1, 3], [2, 1], [2, 3], [3, 1], [3, 2], [3, 4]]$, which is a true subset of $\mathsf{perms}_4^2 = [[1, 2], [1, 3], [1, 4], [2, 1], [2, 3], [2, 4], [3, 1], [3, 2], [3, 4], [4, 1], [4, 2], [4, 3]]$.
- When $m = 2$ and $n = 0$, then the output should be [].
- When $m = 2$ and $n = 2$, then the output should be [[1,2]], which is a true subset of $\mathsf{perms}_2^2$.

## 3 Concepts in the Definition of Isabelle Script

In order to understand the concepts of generalization, it may be better to read an glimpse of a formal proof script. Let us see the definition of this proof scripts at first.

theory n_mutualEx_base imports paraTheory
begin

### 3.1 Definitions of Constants

definition I::”scalrValueType” where [simp]: ”I≡ enum ”control” ”I””
definition T::”scalrValueType” where [simp]: ”T≡ enum ”control” ”T””
definition C::”scalrValueType” where [simp]: ”C≡ enum ”control” ”C””
definition E::”scalrValueType” where [simp]: ”E≡ enum ”control” ”E””
definition true::”scalrValueType” where [simp]: ”true≡ boolV True”
definition false::”scalrValueType” where [simp]: ”false≡ boolV False”

### 3.2 Definitions of Parameterized Rules

definition n_Try::”nat ⇒ rule” where [simp]:
”n_Try i≡

let g = (eqn (IVar (Para (Ident "n") i)) (Const I)) in
let s = (parallelList [(assign ((Para (Ident "n") i), (Const T)))]) in
guard g s"

definition n_Crit::"nat ⇒ rule" where [simp]:
"n_Crit i≡
let g = (andForm (eqn (IVar (Para (Ident "n") i)) (Const T)) (eqn (IVar (Ident "x")) (Const true))) in
let s = (parallelList [(assign ((Para (Ident "n") i), (Const C))), (assign ((Ident "x"), (Const false)))]) in
guard g s"

definition n_Exit::"nat ⇒ rule" where [simp]:
"n_Exit i≡
let g = (eqn (IVar (Para (Ident "n") i)) (Const C)) in
let s = (parallelList [(assign ((Para (Ident "n") i), (Const E)))]) in
guard g s"

definition n_Idle::"nat ⇒ rule" where [simp]:
"n_Idle i≡
let g = (eqn (IVar (Para (Ident "n") i)) (Const E)) in
let s = (parallelList [(assign ((Para (Ident "n") i), (Const I))), (assign ((Ident "x"), (Const true)))]) in
guard g s"

### 3.3 The set of All actual Rules w.r.t. a Protocol Instance with Size *N*

definition rules::"nat ⇒ rule set" where [simp]:
"rules N ≡ {r.(∃ i. i≤N∧r=n_Try i) ∨ (∃ i. i≤N∧r=n_Crit i) ∨
(∃ i. i≤N∧r=n_Exit i) ∨ (∃ i. i≤N∧r=n_Idle i)}"

### 3.4 Definitions of a Formally Parameterized Invariant Formulas

definition inv__1::"nat ⇒ nat ⇒ formula" where [simp]:
"inv__1 p__Inv3 p__Inv4 ≡
(neg (andForm (eqn (IVar (Para (Ident "n") p__Inv4)) (Const C)) (eqn (IVar (Para (Ident "n") p__Inv3)) (Const C))))"

definition inv__2::"nat ⇒ formula" where [simp]:
"inv__2 p__Inv4 ≡
(neg (andForm (eqn (IVar (Para (Ident "n") p__Inv4)) (Const C)) (eqn (IVar (Ident "x")) (Const true))))"

definition inv__3::"nat ⇒ nat ⇒ formula" where [simp]:
"inv__3 p__Inv3 p__Inv4 ≡
(neg (andForm (eqn (IVar (Para (Ident "n") p__Inv3)) (Const C)) (eqn (IVar (Para (Ident "n") p__Inv4)) (Const E))))"

definition inv__4::"nat ⇒ formula" where [simp]:
"inv__4 p__Inv4 ≡
(neg (andForm (eqn (IVar (Para (Ident "n") p__Inv4)) (Const E)) (eqn (IVar (Ident "x")) (Const true))))"

definition inv__5::"nat ⇒ nat ⇒ formula" where [simp]: "inv__5 p__Inv3 p__Inv4 ≡ (neg (andForm (eqn (IVar (Para (Ident "n") p__Inv3)) (Const E)) (eqn (IVar (Para (Ident "n") p__Inv4)) (Const E))))"

### 3.5 Definitions of the Set of Invariant Formula Instances in a $N$-protocol Instance

definition invariants::"nat ⇒ formula set" where [simp]:
"invariants N ≡ {f.
(∃ p__Inv3 p__Inv4. p__Inv3≤N∧p__Inv4≤N∧p__Inv3≠p__Inv4∧f=inv__1 p__Inv3 p__Inv4) ∨
(∃ p__Inv4. p__Inv4≤N∧f=inv__2 p__Inv4) ∨
(∃ p__Inv3 p__Inv4. p__Inv3≤N∧p__Inv4≤N∧p__Inv3≠p__Inv4∧f=inv__3 p__Inv3 p__Inv4) ∨
(∃ p__Inv4. p__Inv4≤N∧f=inv__4 p__Inv4) ∨
(∃ p__Inv3 p__Inv4. p__Inv3≤N∧p__Inv4≤N∧p__Inv3≠p__Inv4∧f=inv__5 p__Inv3 p__Inv4)
}"
definition initSpec0::"nat ⇒ formula" where [simp]:
"initSpec0 N ≡ (forallForm (down N) (% i . (eqn (IVar (Para (Ident "n") i)) (Const I))))"
definition initSpec1::"formula" where [simp]:
"initSpec1 ≡ (eqn (IVar (Ident "x")) (Const true))"
definition allInitSpecs::"nat ⇒ formula list" where [simp]:
"allInitSpecs N ≡ [ (initSpec0 N), (initSpec1 ) ]"

## 4 Generalization of Normalized Concrete Invariant Formulas with Model Constraints

**Definition 1.** *Let $f$ be a concrete formula, we define symbolize$(f)$ to be the formula transformed from $f$ by substituting each concrete parameter $j$ with* $\mathtt{iInv_j}$.

*symbolize*$(f)$ is called the simple symbolic representation of the formula $f$. For instance, let $f$ be $\neg(n[1] \doteq C \barwedge n[2] \doteq C)$, *symbolize*$(f)$ is $\neg(n[iInv_1] \doteq$

$C \barwedge n[iInv_2] \doteq C$). $symbolize(f)$ only shows the syntax effect of symbolic index replacement, we also need model constraints with the symbolic transformation.

**Definition 2.** *Let $N$ be a symbolic value representing a size of a parameterized instance of a protocol, we define:*

1. *model constraint-I: $modelConstrI(N, j) \equiv \mathtt{iInv_j} \leq N$.*
2. *model constraint: $modelConstr(N, L) \equiv forallForm(pf, |L|)$, where $pf(i)$ is $= modelConstrI(N, i)$.*

For instance, $modelConstrI(N, [1]) = \mathtt{iInv_1} \leq N$; $modelConstr(N, [1, 2]) = \mathtt{iInv_1} \leq N \wedge \mathtt{iInv_2} \leq N$. Model constraints intuitively represents that any parameter index should be not greater than $N$, which is the meaning of $N-$paramterized instance.

**Definition 3.** *let $L$ be a permutation,*

1. *difference constraint between parameter $\mathtt{iInv}_i$ and $\mathtt{iInv}_j$: $diff(i, j) \equiv (\mathtt{iInv}_i \neq \mathtt{iInv}_j)$.*
2. *mutual difference constraint: $mutualDiff(L) \equiv \bigwedge S$, where $S$ is a set of HOL formulas, and $S = \{f.f = diff(i, j)$ and $i \leq |L|$ and $j \leq |L|$ and $i < j\}$.*

For instance, $diff(1, 2) = (\mathtt{iInv_1} \neq \mathtt{iInv_2})$; $mutualDiff([1, 2]) = (\mathtt{iInv_1} \neq \mathtt{iInv_2})$; $mutualDiff([1, 2, 3]) = (\mathtt{iInv_1} \neq \mathtt{iInv_2}) \wedge (\mathtt{iInv_1} \neq \mathtt{iInv_3}) \wedge (\mathtt{iInv_2} \neq \mathtt{iInv_3})$. $mutualDiff$ emphasizes that parameters of an formula should be different from each other.

By this generalization, we can easily generate a definition of a formal parameterized invariant formula in Isabelle syntax, which is listed in subsection 3.4.

For instance, $diff(1, 2) = (\mathtt{iInv_1} \neq \mathtt{iInv_2})$; $mutualDiff([1, 2]) = (\mathtt{iInv_1} \neq \mathtt{iInv_2})$; $mutualDiff([1, 2, 3]) = (\mathtt{iInv_1} \neq \mathtt{iInv_2}) \wedge (\mathtt{iInv_1} \neq \mathtt{iInv_3}) \wedge (\mathtt{iInv_2} \neq \mathtt{iInv_3})$. $mutualDiff$ emphasizes that parameters of an formula should be different from each other.

Combining $symbolize$ function with the above two restrictions, we define:

**Definition 4.** *Let $f$ be a normalized concrete formula, 1,2,..n are parameter indice occuring in $f$,*

$$sym(f', f, N) \equiv \begin{cases} f' = f, & n = 0 \quad (1) \\ \begin{array}{c} \exists \mathtt{iInv_1} ... \mathtt{iInv_n}. modelConstr(N, ID_n) \\ \wedge mutualDiff(ID_n) \wedge f' = symbolize(f) \end{array}, & otherwise \quad (2) \end{cases}$$

*$generalizeS([f_1, .., f_m], N) \equiv \{f'.sym(f', f_1, N) \vee ... \vee sym(f', f_m, N)\}$, where $f_i$ is a concrete normalized formula, $ID_n = 1$ upto $n$.*

We call that $f'$ is symmetric to $f$ in the $N-$parameterized protocol instance if $sym(f', f, N)$. The intuition underlying function $sym$ is symmetry. $sym(f', f, N)$ means that there is an index replcaement by which $f'$ can transformed into $f$. For instance, $mutualInv(3, 4)$ satisfies $sym(mutualInv(3, 4), mutualInv(1, 2), 5)$, $mutualInv(3, 4)$ can be transformed into $mutualInv(1, 2)$ by replacing 3 and 4 with 1 and 2 respectively. $generalize(fs, N)$ defines a set of formulas each of which is symmetric to a formula $f_i$ in $fs$.

With the help of generalizeS($[f_1, .., f_m], N$), we can generate the definition of the set of invariant formula instances in a $N$-protocol instance which is listed in 3.5.

# 5 Lemms and Their Generation

## 5.1 An Overview of All the Lemmas

After giving the definitions of rules and invariant formulas, next we show the lemmas and their proofs. First we give the hierarchy of all the Isabelle lemmas generated by proofGen, which is shown in Fig. 5.1, the proof of main lemma needs `invs_on_rules` and `on_inis`. The proof `invs_on_rules` needs `lemma_invi_on_rules`, where $1 \leq i \leq 5$; and the proof of `lemma_invInst_on_rules` such as `lemma_inv1_on_rules` needs a lemma `lemma_invInst_on_ruleInst` such as `critVsinv1`. The proof of Lemma on_inis needs lemma iniImply_inv4. We introduce them one by one in a bottom-up order.

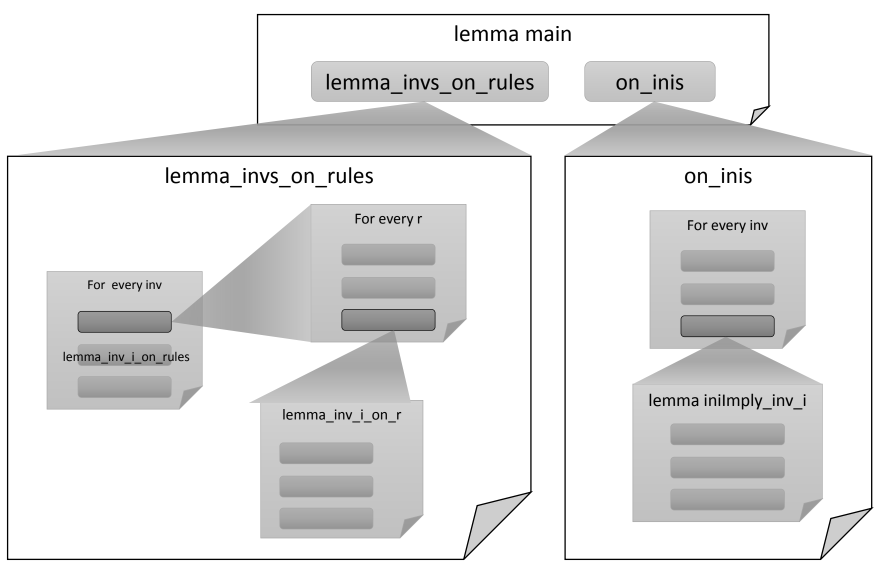


**Fig. 1.** The hierachy of lemmas

**Lemmas for Causal Relation between Rules and Invariants** A lemma at the bottom level, specifies that causal relation hold between a rule like `crit` and a parameterized rule like $inv_1$, An example lemma $\texttt{critVsinv}_1$ and its proof in Isabelle in the `mutualEx` protocol, is illustrated as follows:

```
1lemma critVsinv1:
2 assumes a1: ∃ iR1. iR1 ≤ N ∧ r=crit iR1 and
a2: ∃ iInv1 iInv2. iInv1 ≤ N ∧ iInv2 ≤ N ∧ iInv1 ≠ iInv2 ∧ f=inv1 iInv1 iInv2
3 shows invHoldForRule s f r (invariants N)
4 proof -
from a1 obtain iR1 where a1:iR1 ≤ N ∧ r=crit iR1
  by blast
from a2 obtain iInv1 iInv2 where
a2: iInv1 ≤ N ∧ iInv2 ≤ N ∧ iInv1 ≠ iInv2 ∧ f=inv1 iInv1 iInv2
  by blast
5 have iR1=iInv1 ∨ iR1=iInv2 ∨ (iR1 ≠ iInv1 ∧ iR1 ≠ iInv2) by auto
6 moreover{assume b1:iR1=iInv1
7   have invHoldForRule3 s f r (invariants N)
    proof(cut_tac a1 a2 b1, simp, rule_tac x=! (x=true ⊼ n[iInv2]=C) in exI,auto)qed
8   then have invHoldForRule s f r (invariants N) by auto}
9 moreover{assume b1:iR1=iInv2
10   have invHoldForRule3 s f r (invariants N)
    proof(cut_tac a1 a2 b1, simp, rule_tac x=! (x=true ⊼ n[iInv1]=C in exI,auto)qed
11   then have invHoldForRule s f r (invariants N) by auto}
12 moreover{assume b1:(iR1 ≠ iInv1 ∧ iR1 ≠ iInv2)
13   have invHoldForRule2 s f r
    proof(cut_tac a1 a2 b1, auto) qed
14   then have invHoldForRule s f r (invariants N) by auto}
15ultimately show invHoldForRule s f r (invariants N) by blast
16qed
```

A lemma such as `critVsinv1` is generated by collecting all the records on the invariant `inv1` and rule `crit` in the aforementioned tables. Line 2 are assumptions on the parameters of the invariant and rule, which are composed of two parts: (1) assumption `a1` specifies that there exists an actual parameter `iR1` with which `r` is a rule obtained by instantiating `crit`; (2) assumption `a2` specifies that there exists actual parameters `iInv1` and `iInv2` with which `f` is a formula obtained by instantiating `inv1`. Line 4 are two typical proof patterns forward-style which fixes local variables such as `iR1` and new facts such as `a1: iR1` $\leq$ `N` $\wedge$ `r=crit iR1`. From line 5, the remaining parts of the proof is a typically readable one in Isar style [**?**], which uses calculation reasoning such as `moreover` and `ultimately` to do case analysis. Line 5 splits cases of `iR1` into all possible cases by comparing `iR1` with `iInv1` and `iInv2`, which is in fact characterized by $partition([1],[2],[3]],[1,2])$. Lines 6-14 proves these cases one by one: Lines 6-8 proves the case where `iR1=iInv1`, line 7 first proves that the causal relation $invHoldForRule_3$ holds by supplying a formula, which is $symbolize'(invOnXC(2),[1,2],[1])$. From the conclusion at line 7, line 8 futhermore proves the causal relation $invHoldForRule$ hold; Lines 9-11 proves the case where `iR1=iInv2`, proof of which is similar to that of case 1; Lines 12-14 the case where neither `iR1=iInv1` nor `iR1=iInv2`. Each proof of a subcase is done in a block `moreover b1:asm1 proof1`, the `ultimately` proof command in line 15 concludes by summing up all the subcases.

With the help of all the lemmas such as ruleVsinv1, we can prove the following lemma lemma_inv_1_on_rules which specifies that for all $r \in rules\ N$, and $f$ is a formula $f$ which is generated by instantiating inv1 with some parameters $iInv_1$ and $iInv_2$, $invHoldForRule\ s\ f\ r\ (invariants\ N)$.

```
lemma lemma_inv1_on_rules: ⟦ a1: r ∈ rules N and a2: (∃ _iInv1 _iInv2. _iInv1≤ N∧_iInv2≤
N∧iInv1≠iInv2∧f=inv1 iInv1 iInv2)
⟧ ⟹ invHoldForRule s f r (invariants N)
proof -
  have (∃ i. i≤ N∧r=try i)∨ (∃ i. i≤
N∧r=crit i)∨ (∃ i. i≤ N∧r=exit i)∨
(∃ i. i≤ N∧r=idle i)
apply (cut_tac a1, auto) done
moreover { assume b1: (∃ i. i≤ N∧r=try i)
  have invHoldForRule' s f r (invariants N)
  apply (cut_tac a2 b1, metis tryVsinv1) done }
moreover { assume a1: (∃ i. i≤ N∧r=crit i)
  have invHoldForRule' s f r (invariants N)
  apply (cut_tac a2 b1, metis critVsinv1) done }
moreover { assume a1: (∃ i. i≤ N∧r=exit i)
  have invHoldForRule' s f r (invariants N)
  apply (cut_tac a2 b1, metis exitVsinv1) done }
moreover { assume a1: (∃ i. i≤ N∧r=idle i)
  have invHoldForRule' s f r (invariants N)
  apply (cut_tac a2 b1, metis idleVsinv1) done }
ultimately show invHoldForRule' s f r (invariants N)
by auto
qed
```

With the help of all the lemmas such as $\mathsf{lemma\_inv}_i\mathsf{\_on\_rules}$, we can prove the following lemma invs_on_rules which specifies that for all $f \in invariants\ N$ and $r \in rules\ N$, $invHoldForRule\ s\ f\ r\ (invariants\ N)$.

```
lemma invs_on_rules: ⟦ a1: f ∈ invariants N and a2: r ∈ rules N ⟧ ⟹
invHoldForRule' s f r (invariants N)
proof -
have b1: (∃ iInv1 iInv2. iInv1≤ N∧iInv2≤ N∧iInv1≠iInv2∧f=inv1 iInv1 iInv2)∨
(∃ iInv2. iInv2≤ N∧f=inv2 iInv2)∨
(∃ iInv1 iInv2. iInv1≤ N∧iInv2≤ N∧iInv1≠iInv2∧f=inv3 iInv1 iInv2)∨
(∃ iInv2. iInv2≤ N∧f=inv4 iInv2)∨
(∃ iInv1 iInv2. iInv1≤ N∧iInv2≤ N∧iInv1≠iInv2∧f=inv5 iInv1 iInv2)
apply (cut_tac a1, auto) done
moreover { assume b1: (∃ iInv1 iInv2. iInv1≤ N∧iInv2≤ N∧iInv1≠iInv2∧f=inv1 iInv1 iInv2)
  have invHoldForRule' s f r (invariants N)
  apply (cut_tac a2 b1, metis lemma_inv1_on_rules) done }
moreover { assume b1: (∃ iInv2. iInv2≤ N∧f=inv2 iInv2)
  have invHoldForRule' s f r (invariants N)
  apply (cut_tac a2 b1, metis lemma_inv2_on_rules) done }
moreover { assume b1: (∃ iInv1 iInv2. iInv1≤ N∧iInv2≤ N∧iInv1≠iInv2∧f=inv3 iInv1 iInv2)
  have invHoldForRule' s f r (invariants N)
  apply (cut_tac a2 b1, metis lemma_inv3_on_rules) done }
moreover { assume b1: (∃ iInv2. iInv2≤ N∧f=inv4 iInv2)
  have invHoldForRule' s f r (invariants N)
  apply (cut_tac a2 b1, metis lemma_inv4_on_rules) done }
moreover { assume b1: (∃ iInv1 iInv2. iInv1≤ N∧iInv2≤ N∧iInv1≠iInv2∧f=inv5 iInv1 iInv2)
  have invHoldForRule' s f r (invariants N)
  apply (cut_tac a2 b1, metis lemma_inv5_on_rules) done }
ultimately show invHoldForRule' s f r (invariants N)
  apply fastforce done
qed end
```

**Lemmas on initial states** In this section, we discuss the definition on the initial state of the protocol, and the lemmas specifying that each invariant formula holds at the initial state.

A typical Isabelle definition on the initial state of the protocol is as follows:

```
definition initSpec0::nat ⇒ formula where [simp]:
initSpec0 N ≡ (forallForm (down N) (% i . (eqn (IVar (Para (Ident ''n'') i)) (Const I))))
definition initSpec1::formula where [simp]:
initSpec1 ≡ (eqn (IVar (Ident ''x'')) (Const true))
definition allInitSpecs::nat Rightarrow> formula list where [simp]:
allInitSpecs N ≡ [(initSpec0 N),(initSpec1 )]
lemma iniImply_inv4: assumes a1: (∃iInv1. iInv1≤N∧f=inv4 iInv1)
and a2: formEval (andList (allInitSpecs N)) s
shows formEval f s
using a1 a2 by auto
```

`initSpec0` and `initSpec1` specifies the assignments on each variable `n[i]` where `i` $\leq$ `N` and `x`. The specifications of the initial state is the list of all the specification definition on related state variables. Lemma `iniImply_inv4` simply specifies that the invariant formula `inv4` holds at a state `s` which satisfies the conjunction of the specification of the initial state. Isabelle's `auto` method can solve this goal automatically. Other lemmas specifying that other invariant formulas hold at the initial state are similar.

With the lemmas such as `iniImply_inv4`, for any invariant $inv \in (\mathsf{invariants}\ N)$, any state $s$, if $ini$ is evaluated true at state $s$, then $inv$ is evaluated true at state $s$.

```
lemma on_inis: ⟦ a1: f ∈ (invariants N) and a2: ini ∈ { andList (allInitSpecs N)}
and a3: formEval ini s ⟧ ⟹ formEval f s
proof -
have c1: (∃ iInv1 iInv2. iInv1≤ N∧iInv2≤ N∧iInv1≠iInv2∧f=inv__1 iInv1 iInv2)∨
(∃ iInv2. iInv2≤ N∧f=inv__2 iInv2)∨
(∃ iInv1 iInv2. iInv1≤ N∧iInv2≤ N∧iInv1≠iInv2∧f=inv__3 iInv1 iInv2)∨
(∃ iInv2. iInv2≤ N∧f=inv__4 iInv2)∨
(∃ iInv1 iInv2. iInv1≤ N∧iInv2≤ N∧iInv1≠iInv2∧f=inv__5 iInv1 iInv2)
  apply (cut_tac a1, simp) done
moreover { assume b1: (∃ iInv1 iInv2. iInv1≤ N∧iInv2≤ N∧iInv1≠iInv2∧f=inv__1 iInv1 iInv2)
have formEval f s
  apply (rule iniImply_inv__1)
  apply (cut_tac b1, assumption)
  apply (cut_tac a2 a3, blast) done }
moreover { assume b1: (∃ iInv2. iInv2≤ N∧f=inv__2 iInv2)
have formEval f s
  apply (rule iniImply_inv__2)
  apply (cut_tac b1, assumption)
  apply (cut_tac a2 a3, blast) done
}
moreover { assume b1: (∃ iInv1 iInv2. iInv1≤ N∧iInv2≤ N∧iInv1≠iInv2∧f=inv__3 iInv1 iInv2)
have formEval f s
  apply (rule iniImply_inv__3)
  apply (cut_tac b1, assumption)
  apply (cut_tac a2 a3, blast) done }
moreover { assume b1: (∃ iInv2. iInv2≤ N∧f=inv__4 iInv2)
have formEval f s
  apply (rule iniImply_inv__4)
  apply (cut_tac b1, assumption)
  apply (cut_tac a2 a3, blast) done }
moreover { assume b1: (∃ iInv1 iInv2. iInv1≤ N∧iInv2≤ N∧iInv1≠iInv2∧f=inv__5 iInv1 iInv2)
have formEval f s
  apply (rule iniImply_inv__5)
  apply (cut_tac b1, assumption)
  apply (cut_tac a2 a3, blast) done }
ultimately show formEval f s by auto
qed
```

The proof structure of lemma_inv1_on_rules and invs_on_rules and on_inis are also typical case analysis ones using moreover blocks and ultimately commands, therefore, a generic program of generating a typical case analysis proof will be adopted in our framework.

**The main theorem** With the preparation of lemma on_inis and invs_on_rules, the generation of the main lemma is quite easy. Recall that the consistency lemma is our main weapon to prove the main lemma, which requires proving two parts of obligations.

**(1)** For any invariant $inv \in$ (invariants $N$), any state $s$, if $ini$ is evaluated true at state $s$, then $inv$ is evaluated true at state $s$. This can be solved done by applying lemma on_inis.

**(2)** For any invariant $inv \in$ (invariants $N$), any $r$ in rule set rules $N$ , one of the causal relations invHoldForRule$_{1-3}$ holds. This can be solved done by applying lemma invs_on_rules.

```
lemma main: ⟦ s∈ reachableSet {andList (allInitSpecs N)} (rules N); 0<N⟧
⟹ ∀ inv. inv ∈ (invariants N) ⟶ formEval inv s
proof(rule consistentLemma)
show consistent (invariants N) {andList (allInitSpecs N)} (rules N)
proof(cut_tac a1, unfold consistent_def,rule conjI)
show ∀inv ini s. inv ∈ (invariants N) ⟶ ini ∈{andList (allInitSpecs N)}⟶formEval ini s ⟶ formEval inv s
proof((rule allI)+,(rule impI)+)
  fix inv ini s
  assume b1:inv ∈ (invariants N)
  and b2:ini ∈ {andList (allInitSpecs N)} and b3:formEval ini s
  show "formEval f s"
  apply (rule on_inis, cut_tac b1, assumption, cut_tac b2, assumption, cut_tac b3, assumption) done
qed
next show ∀inv r. inv ∈ invariants N⟶ r ∈rules N⟶invHoldForRule inv r (invariants N)
proof((rule allI)+,(rule impI)+)
  fix f r
  assume b1: f ∈ invariants N and b2:r ∈ rules N
  show "invHoldForRule' s f r (invariants N)"
apply (rule invs_on_rules, cut_tac b1, assumption, cut_tac b2, assumption) done
qed
next show "s ∈ reachableSet andList (allInitSpecs N) (rules N)" apply (metis a1) done
qed
```

The generation of the main lemma is quite easy because it is in a standard form.

### 5.2 Algorithms of Proof Generator proofGen

In this subsection, we illustrate the key techniques and algorithms of generation of the lemmas and their proofs in subsection **??**. Being according with the order in which we introduce the above lemmas, we also introduce their generation in a bottom-up order. First let us introduce the generation of a subproof according to a relation tag of $invHoldForRule_{1-3}$, which is shown in Algorithm 2.

In the body of function rel2proof, $sprintf$ writes a formatted data to string and returns it. In line 10, $getFormField(relTag)$ returns $f'$ if $relTag = invHoldForRule_3(f')$. rel2proof transforms a a relation tag into a paragraph of proof.If the tag is among

**Algorithm 2:** Generating a kind of proof which is according with a relation tag of $invHoldForRule_{1-3}$ : rel2proof

**Input**: A causal relation item $relTag$
**Output**: An Isablle proof: $proof$

**if** $relTag = invHoldForRule_1$ **then**
  $proof \leftarrow$ sprintf
    "have invHoldForRule1 f r (invariants N)
    by(cut_tac a1 a2 b1, simp, auto)
    then have invHoldForRule f r (invariants N) by blast" ;
**else if** $relTag = invHoldForRule_2$ **then**
  $proof \leftarrow$ sprintf
    "have invHoldForRule2 f r (invariants N) by(cut_tac a1 a2 b1, simp, auto)
    then have invHoldForRule f r (invariants N) by blast" ;
**else**
  $f' \leftarrow getFormField(relTag)$;
  $proof \leftarrow$ sprintf
    "have invHoldForRule3 f r (invariants N)
    proof(cut_tac a1 a2 b1, simp, rule_tac x=%s in exI,auto)qed
    then have invHoldForRule f r (invariants N) by blast" (symbf2Isabelle f')";
**return** $proof$

$invHoldForRule_{1-2}$, the transformation is rather straight-forward, else the form $f'$ is assigned by the formula $getFormField(relTag)$, and provided to tell Isabelle the formula which should be used to construct the $invHoldForRule_3$ relation.

**Algorithm 3:** Generating one sub-proof for a subcase: oneMoreOverGen

**Input**: A formula $caseFsm$ standing for the assumption of the subcase, a relation item $relItem$ containing the information of causal relation
**Output**: An Isablle proof: $subProof$

$proof \leftarrow rel2proof(relItem)$;
$subProof \leftarrow$ sprintf
  "moreover{assume b1:%s
%s } "
  ( asm, proof);
**return** $subproof$

In Algorithm 3, oneMoreOverGen generates a subproof for a subcase in a proof of case analysis. It returns a subproof which is composed by filling an assumption of the subcase such as "iR1=iInv1" and a paragraph of proof generated by $rel2proof(relItem)$ into a format of block `morover { ...}`.

Due to the common use of case analysis proof of using moreover and ultimately commands, we design a generic program of generating doing case analysis doCaseAnalz. In algorithm 4, formulas standing for case-splitting $partition$, subproofs $subproofs$, and the conclusion *concluding* are needed in case analysis to fill the format.

**Algorithm 4:** Generating a whole proof of doing case analysis: doCaseAnalz

**Input**: A formula *partition* standing for case-splittings, a proof list $subproofs$ standing all the subproofs of each subcases, concluding parts *concluding*

**Output**: An Isablle proof: $proof$

$proof$ ←sprintf
  " have %s by auto
  %s
  ultimately show %s by auto"
 (partition, subproofs, concluding) ;
**return** $proof$

In algorithm 5, caseAnalzI generates a typical proof of doing case analysis to prove some causal relation hold between some rule and invariant. oneMoreOverGenI(case,rel) formula comes from the disjunction of formulas in the `symbCases` field of $rec$, which is returned by $caseField(rec)$, subproofs $subproofs$ are generated by concatenation of all the subproofs, each of which is generated by $oneMoreOverGenI(case, rel)$. The proof is simply composed by calling $doCaseAnalz(partition, subproofs, concluding)$.

**Algorithm 5:** Generating a whole proof of doing case analysis on parameters of rule and invariant: caseAnalzI

**Input**: A record $rec$ fetched from $symbCausal$

**Output**: An Isablle proof: $proof$

$cases \leftarrow caseField(rec)$;
$rels \leftarrow relItems(rec)$; $partition \leftarrow \bigvee cases$;
$subproofs \leftarrow$ "";
**while** ($cases \neq []$) **do**
  $case \leftarrow hd(cases)$ ;
  $cases \leftarrow tl(cases)$ ;
  $rel \leftarrow hd(rels)$ ;
  $rels \leftarrow tl(rels)$ ;
  $subproofs \leftarrow subproofs\hat{\ }oneMoreOverGenI(case, rel)$;
$concluding$ ←"invHoldForRule s f r (invariants N) ";
$proof \leftarrow doCaseAnalz(partition, subproofs, concluding)$;
**return** $proof$

Next we discuss how to generate assumptions on an invariant formula of an lemma such as $critVsInv1$. In the body of algorithm 6, $tbl\_element(symbInvs, invName)$ retrieves the record on a invariant formula from $symbInvs$ to $invItem$ by its name $invName$, $invParaNum(invItem)$ and $constrOfInv(invItem))$ return the field `invNumFld` and `constr` of $invItem$ respectively. $invParasGen(lenPInv)$ generates a string of a list of actual parameters such as $iInv_1...iInv_{lenPInv}$ if $lenPInv > 0$, else an empty string "". At last, the assumption on the invariant is created by filling $invParas$, $constrOnInv$, and $invName$ into a proper place in the format if needed.

---

**Algorithm 6:** Generating an assumption on an invariant formula: asmGenOnInv

---

**Input**: An invariant name $invName$, a table $symInvs$ storing invariant formulas
**Output**: An assumption on an invariant formula: $asm$

```
invItem ← tbl_element(symbInvs, invName);
lenPInv ← invParaNum(invItem);
invParas ← invParasGen(lenPInv);
constrOnInv ← symbForm2Isabelle(constrOfInv(invItem));
if lenPInv = 0 then
    asm ← "a1 : f = "^invName;
else
    asm ← sprintf "a1: ∃ %s. %s ∧ f=%s %s" (invParas, constrOnInv, invName,
    invParas);
return asm
```

---

Similar to asmGenOnInv, obtainGenOnInv, which is shown in algorithm 7, generates a proof command of obtain by retrieving and generating the related information and filling them in a format on obtain. Similar to asmGenOnInv and obtainGenOnInv, asmGenOnRule and obtainGenOnRule generate an assumption and obtain proof command on a rule.

---

**Algorithm 7:** Generating an obtain proof command on an invariant formula: obtainGenOnInv

---

**Input**: An invariant name $invName$, a table $symInvs$ storing rules

```
invItem ← tbl_element(symbInvs, invName);
lenPInv ← invParaNum(invItem);
invParas ← invParasGen(lenPInv);
if lenPInv = 0 then
    obtain ← "";
else
    obtain ← sprintf "from a1 obtain %s where a1:%s ∧ f=%s %s by auto"
      (invParas, constrOnInv, invName, invParas);
return obtain
```

---

After the above preparing functions, now the generation of a lemma on the causal relation such as $critVsInv1$ is rather easy, which is shown in algorithm 8.

After generating an assumption on invariant formula $asm1$, $asm2$ on a rule, an obtain command $obtain1$ on the invariant, and $obtain2$ on the rule, $symRelItem$ is retrieved from $symCausalTab$ by $ruleName\hat{}invName$, and a proof $proof$ is generated by calling $caseAnalzI(symRelItem)$. At last these parts are filled into proper places in the lemma format.

---

**Algorithm 8:** Generating a lemma on a causal relation: lemmaOnCausalRuleInv

---

**Input**: A parameterized rule name $ruleName$, a formula name $invName$, a table $symRules$ storing rules , a table $symInvs$ storing invariant formulas, a table $symCausalTab$ storing causal relation

**Output**: An Isablle proof script for a lemma: $lemmaWithProof$

$asm1 \leftarrow asmGenOnInv(symbInvs, invName)$;<br>
$asm2 \leftarrow asmGenOnRule(symbRules, ruleName)$;<br>
$obtain1 \leftarrow obtainGenOnInv(symbInvs, invName)$;<br>
$obtain2 \leftarrow obtainGenOnRule(symbRules, ruleName)$;<br>
$symRelItem \leftarrow tbl\_element(symCausalTab, (ruleName\hat{}invName))$;<br>
$proof \leftarrow caseAnalzI(symRelItem)$;<br>
$lemmaWithProof \leftarrow$ sprintf<br>
  "lemma %sVs%s:<br>
  assumes %s and %s<br>
  shows invHoldForRule s f r (invariants N)<br>
  proof -   %s  %s  %s<br>
  qed"<br>
  $(ruleName, invName, asm1, asm2, obtain1, obtain2, proof)$ ;<br>
**return** $lemmaWithProof$

---