Semantics of the SELinux policy Language in $\mathbb{K}$

1 Introduction

This document is for the core implementation of K-SELinux, including semantics and several formal analysis of security properties written in $\mathbb{K}$ framework.

2 Syntax part

This part is for the syntax of statements in the SELinux policy language.

```
module MLS-SYNTAX
    imports ID

    syntax LevelId ::= Id
    syntax MlsRange ::= LevelId
                      | LevelId "-" LevelId
endmodule

module SELINUX-SYNTAX-CORE
    imports MLS-SYNTAX
    imports BOOL
    imports ID
    imports STRING
    imports UNSIGNED-INT-SYNTAX

    syntax KItem ::= "(" TypeId   TypeId   ClassId ")"
                   | "(" TypeId   WildCard ClassId ")"
                   | "(" WildCard TypeId   ClassId ")"
                   | "(" WildCard WildCard ClassId ")"

    syntax Name ::= Id

    syntax ClassId  ::= Id
    syntax CommonId ::= Id
    syntax PermId   ::= Id
    syntax TypeId   ::= Id
    syntax AttrId   ::= Id
    syntax AliasId  ::= Id
    syntax UserId   ::= Id
    syntax RoleId   ::= Id
    syntax BoolId   ::= Id
    syntax SidId    ::= Id
    syntax CapId    ::= Id

    syntax Context ::= UserId ":" RoleId ":" TypeId
                     | UserId ":" RoleId ":" TypeId ":" LevelId

    syntax SidStmt ::= "sid" SidId
                     | "sid" SidId Context

    syntax PolicycapStmt ::= "policycap" CapId ";"

    syntax ClassPermPair ::= "(" ClassId PermId ")"

    // name rules
    // XXXIds ::= List{XXXId, ""}
    // XXXs     ::= XXXId | "{" XXXIds "}"
    // XXXList  ::= List{XXXId, ","}

    syntax BoolList ::= List{BoolId, ","}

    syntax ClassIds ::= List{ClassId, ""}
    syntax Classes ::= ClassId | "{" ClassIds "}"

    syntax PermIds ::= List{PermId, ""}
    syntax Perms ::= PermId | "{" PermIds "}"
```

```
syntax TypeIds ::= List{TypeId, ""}
syntax Types ::= TypeId | "{" TypeIds "}"

syntax DashTypeId ::= TypeId | "-" TypeId
syntax DashTypeIds ::= List{DashTypeId, ""}
syntax DashTypes ::= DashTypeId | "{" DashTypeIds "}"

syntax AliasIds ::= List{AliasId, ""}
syntax Aliases ::= AliasId | "{" AliasIds "}"

syntax TypeList ::= List{TypeId, ","}
syntax AttrList ::= List{AttrId, ","}
syntax UserList ::= List{UserId, ","}
syntax RoleList ::= List{RoleId, ","}

syntax RoleIds ::= List{RoleId, ""}
syntax Roles ::= RoleId | "{" RoleIds "}"


syntax BoolDeclStmt ::= "bool" BoolId Bool ";"

// syntax LogicalOp ::= "&&" | "||" | "^" | "==" | "!="

syntax CondExp ::= BoolId
                 | "(" CondExp ")" [bracket]
                 > "!" CondExp
                 > CondExp "&&" CondExp  [left]
                 > CondExp "||" CondExp  [left]
                 > CondExp "==" CondExp  [left]
                 | CondExp "!=" CondExp  [left]

syntax BoolOp ::= "==" | "!=" | "eq" | "dom" | "domby" | "incomp"
syntax BoolItem ::= Name BoolOp Name

syntax ConstraintExp ::= BoolItem
                       | "(" ConstraintExp ")"              [bracket]
                       > ConstraintExp "and" ConstraintExp  [left]
                       > ConstraintExp "or"  ConstraintExp  [left]


syntax CommonDeclStmt ::= "common" CommonId "{" PermIds "}"

syntax ClassDeclStmt ::= "class" ClassId

syntax ClassAccessStmt ::= "class" ClassId "{" PermIds "}"
                         | "class" ClassId "inherits" CommonId
                         | "class" ClassId "inherits" CommonId "{" PermIds "}"

syntax AttrDeclStmt ::= "attribute" AttrId ";"

syntax TypeDeclStmt ::= "type" TypeId ";"
                      | "type" TypeId "alias" Aliases ";"
                      | "type" TypeId "," AttrList ";"
                      | "type" TypeId "alias" Aliases "," AttrList ";"

syntax TypeAttrStmt ::= "typeattribute" TypeId AttrList ";"

syntax TypeAliasStmt ::= "typealias" TypeId "alias" Aliases ";"

syntax TypeTransition ::= "type_transition" DashTypes DashTypes ":" Classes TypeId ";"
                        | "type_transition" DashTypes DashTypes ":" Classes TypeId String ";
                          "


syntax TypeChangeStmt ::= "type_change" Types Types ":" Classes TypeId ";"
syntax TypeMemberStmt ::= "type_member" Types Types ":" Classes TypeId ";"

syntax PermissiveStmt ::= "permissive" TypeId ";"

// user and role
syntax RoleDeclStmt ::= "role" RoleId ";"
                      | "role" RoleId "types" Types ";"

syntax AttrRoleStmt ::= "attribute_role" AttrId ";"

syntax RoleAttrStmt ::= "roleattribute" RoleId AttrId ";"

syntax RoleAllowStmt ::= "allow" Roles Roles ";"

syntax RoleTransitionStmt ::= "role_transition" Roles Types RoleId ";"
                            | "role_transition" Roles Types ":" ClassId RoleId ";"

syntax RoleItem ::= "role" RoleId
syntax RoleItems ::= List{RoleItem, ";"}
```

```
syntax RoleDominance ::= RoleItems "{" RoleDominance "}"
                       | RoleItems ";"

syntax RoleDomStmt ::= "dominance" "{" RoleDominance "}"
syntax UserDeclStmt ::= "user" UserId "roles" Roles ";"
                      | "user" UserId "roles" Roles "level" LevelId "range" MlsRange ";"

// constrain statement
syntax ConstrainStmt ::= "constrain" Classes Perms "(" ConstraintExp ")" ";"
syntax ValidatetransStmt ::= "validatetrans" Classes ConstraintExp ";"


// if statement
syntax IfStmt ::= "if" "(" CondExp ")" "{" StmtList "}"
                | "if" "(" CondExp ")" "{" StmtList "}" "else" "{" StmtList "}"

syntax OptionalStmt ::= "optional" "{" StmtList "}"


// access vector
syntax AccessRule ::= "allow"         [token]
                    | "auditallow"    [token]
                    | "auditdeny"     [token]
                    | "dontaudit"     [token]
                    | "neverallow"    [token]

syntax WildCard ::= "*" [token]

syntax TypeItem ::= TypeId | "-" TypeId
syntax TypeItems ::= List{TypeItem, ""}
syntax TypeItemSet ::= TypeItem
                     | "~" TypeItem
                     | "{" TypeItems "}"
                     | "~" "{" TypeItems "}"
                     | WildCard

syntax ClassItem ::= ClassId // | "-" ClassName
syntax ClassItems ::= List{ClassItem, ""}
syntax ClassItemSet ::= ClassItem
                      | "{" ClassItems "}"
                      | "~" "{" ClassItems "}"
                      | WildCard

// syntax PermItem ::= PermId | "-" PermId
// syntax PermItems ::= List{PermItem, ""}
// syntax PermItemSet ::= PermItem
//                      | "~" PermItem
//                      | "{" PermItems "}"
//                      | "~" "{" PermItems "}"
//                      | WildCard
syntax PermItem ::= PermId
syntax PermItems ::= List{PermItem, ""}
syntax PermItemSet ::= PermItem
                     | "~" PermItem
                     | "{" PermItems "}"
                     | "~" "{" PermItems "}"
                     | WildCard

syntax AccessVector ::= AccessRule TypeItemSet TypeItemSet ":" ClassItemSet PermItemSet

syntax Object ::= TypeId ":" ClassId

syntax XpermRule ::= "allowxperm"       [token]
                   | "dontauditxperm"   [token]
                   | "auditallowxperm"  [token]
                   | "neverallowxperm"  [token]

syntax XpermOperation ::= Id
syntax XpermRuleStmt ::= XpermRule TypeItemSet TypeItemSet ":" ClassItemSet XpermOperation
    PermItemSet ";"

// require statement
syntax RequireStmt ::= "require" "{" RequiredList "}"
                     | "require" RequiredStmt

syntax RequiredList ::= List{RequiredStmt, ""}
syntax RequiredStmt ::= RequiredClassStmt
                      | RequiredRoleStmt
                      | RequiredAttrStmt
                      | RequiredUserStmt
                      | RequiredSensStmt
                      | RequiredCategoryStmt
                      | RequiredTypeStmt
                      | RequiredBoolStmt
```

```
                    | RequiredAttrRoleStmt

syntax RequiredClassStmt ::= "class" ClassId ";"
                           | "class" ClassId Perms ";"

syntax RequiredTypeStmt ::= "type" TypeList ";"
syntax RequiredAttrStmt ::= "attribute" AttrList ";"
syntax RequiredUserStmt ::= "user" UserList ";"
syntax RequiredRoleStmt ::= "role" RoleList ";"
syntax RequiredBoolStmt ::= "bool" BoolList ";"
syntax RequiredAttrRoleStmt ::= "attribute_role" AttrId ";"
// syntax RequiredSensStmt ::= "sensitivity" SensList ";"
// syntax RequiredCategoryStmt ::= "category" CategoryList ";"

syntax FsName ::= Id
syntax NonRootPath ::= List{Id, "/"}
syntax PartialPath ::= "/" NonRootPath
syntax GenfsconStmt ::= "genfscon" FsName PartialPath Context

syntax FsUseTransStmt ::= "fs_use_trans" FsName Context ";"
syntax FsUseXattrStmt ::= "fs_use_xattr" FsName Context ";"
syntax FsUseTaskStmt  ::= "fs_use_task"  FsName Context ";"

// syntax UInt ::= r"[0-9]+" [prefer, token, prec(3)]

syntax Protocol ::= "tcp" | "udp" | "sctp"
syntax PortNumber ::= Int | Int "-" Int
syntax PortconStmt ::= "portcon" Protocol PortNumber Context

syntax ExpandAttrStmt ::= "expandattribute" AttrId Bool ";"

syntax NetworkLabelStmt ::= NetifconStmt
                          | NodeconStmt
                          | PortconStmt

syntax FsLabelStmt ::= FsUseTaskStmt
                     | FsUseTransStmt
                     | FsUseXattrStmt
                     | GenfsconStmt

syntax BoundsRuleStmt ::= TypeboundsStmt

syntax PolicyConfStmt ::= PolicycapStmt

syntax MlsStmt ::= SensitivityStmt
                 | DominanceStmt
                 | CategoryStmt
                 | LevelStmt
                 | RangeTransStmt
                 | MlsconstrainStmt
                 | MlsvalidatetransStmt

syntax ModularPolicyStmt ::= ModuleStmt
                           | RequireStmt
                           | OptionalStmt

syntax UserStmt ::= UserDeclStmt

syntax RoleStmt ::= RoleDeclStmt
                  | AttrRoleStmt
                  | RoleAttrStmt
                  | RoleAllowStmt
                  | RoleTransitionStmt
                  | RoleDomStmt

syntax TypeStmt ::= TypeDeclStmt
                  | AttrDeclStmt
                  | TypeAliasStmt
                  | TypeAttrStmt
                  | TypeTransition
                  | TypeChangeStmt
                  | TypeMemberStmt
                  | PermissiveStmt

syntax ObjectClassStmt ::= ClassDeclStmt
                         | ClassAccessStmt
                         | CommonDeclStmt

syntax ConstraintStmt ::= ConstrainStmt
                        | ValidatetransStmt

syntax AccessVectorRuleStmt ::= AccessVector ";"

syntax ConditionalPolicyStmt ::= BoolDeclStmt
```

```
                                | IfStmt

    syntax Stmt ::= RoleStmt
                  | UserStmt
                  | TypeStmt
                  | ObjectClassStmt
                  | ConstraintStmt
                  | AccessVectorRuleStmt
                  | ConditionalPolicyStmt
                  | ModularPolicyStmt
                  | MlsStmt
                  | SidStmt
                  | PolicyConfStmt
                  | FsLabelStmt
                  | NetworkLabelStmt
                  | BoundsRuleStmt
                  | DefaultObjectRuleStmt
                  | XenStmt
                  | XpermRuleStmt
                  | ExpandAttrStmt
                  | "{" StmtList "}"
                  | ";"

    syntax StmtList ::= List{Stmt, ""}

    syntax Pgm ::= List{Stmt, ""}
endmodule

module SELINUX-SYNTAX
    imports SELINUX-SYNTAX-CORE
endmodule
```

3 Configuration part

This part is for the global execution model of the SELinux policy language, written as configurations in $\mathbb{K}$ framework.

```
module CONFIGURATION
    imports SELINUX-SYNTAX
    imports SET
    imports MAP
    imports LIST
    imports INT
    imports ID

    // syntax Pgm ::= List{Stmt, ""}
    syntax Controller ::= "check"
                        | "all-last"

    configuration <T>
        <k> $PGM:Pgm ~> check ~> all-last </k>
        // <k> $PGM:Pgm ~> all-end </k>

        // class
        <classes>       .Set </classes>
        <commons>       .Set </commons>
        <common-perms>  .Map </common-perms>
        <class-perms>   .Map </class-perms>

        // type
        <types>      .Set </types>
        <attrs>      .Set </attrs>
        <type-alias> .Map </type-alias>
        <attr-types> .Map </attr-types>

        // sid
        <sids> .Map </sids>
        <caps> .Set </caps>

        // access
        <allow>      .Set </allow>
        // <auditallow> .Map </auditallow>
        // <dontaudit>  .Map </dontaudit>
        // <auditdeny>  .Map </auditdeny>
        <neverallow> .Set </neverallow>
        <neverallow-map> .Map </neverallow-map>
        <wrongallow> .Set </wrongallow>

        // constrain
        <constraints>   .Map </constraints>
        <validatetrans> .Map </validatetrans>
```

```
        <constraint-failed> .Set </constraint-failed>

        // role
        <roles>      .Set </roles>
        <users>      .Set </users>
        <attr-roles> .Map </attr-roles>
        <user-roles> .Map </user-roles>
        <role-types> .Map </role-types>
        <role-allow> .Map </role-allow>
        <role-trans> .Set </role-trans>
        <dominance>  .Set </dominance>

        // mls
        <sensitivities>    .Set </sensitivities>
        <sens-alias>       .Map </sens-alias>
        <levels>           .Map </levels>
        <categories>       .Set </categories>
        <dominances>       .Set </dominances>
        <category-alias>   .Map </category-alias>
        <mlsconstrains>    .Map </mlsconstrains>

        // condition
        <bools> .Map </bools>

        // require failed
        <require-failed> .Set </require-failed>

        // to verify
        <class-remained> .Set </class-remained>
        <perm-remained>  .Map </perm-remained>
        <type-remained>  .Set </type-remained>

        <rules>   .List </rules>
        <domains> .Set </domains>
        <objects> .Set </objects>

        <domain-labels>
            <D-read>  .Map </D-read>
            <D-write> .Map </D-write>
        </domain-labels>

        <object-type-labels>
            <O-read>  .Map </O-read>
            <O-write> .Map </O-write>
        </object-type-labels>
        // <trusted-type> SetItem(type root:Id ;) </trusted-type>

        // consistency
        <indirect-rules> .Set </indirect-rules>
        <indirect-rules-result> .Set </indirect-rules-result>

        // coverage
        <object-noread>  .Set </object-noread>
        <object-nowrite> .Set </object-nowrite>
        <domain-noread>  .Set </domain-noread>
        <domain-nowrite> .Set </domain-nowrite>
        <t1-unused> .Set </t1-unused>
        <t2-unused> .Set </t2-unused>

        <perms-used> .Map </perms-used>
        <perms-unused> .Map </perms-unused>

        <count>
            <bools-count> 0 </bools-count>
            <classes-count> 0 </classes-count>
            <types-count> 0 </types-count>
            <allow-count> 0 </allow-count>
            <neverallow-count> 0 </neverallow-count>
            <domains-count> 0 </domains-count>
            <objects-count> 0 </objects-count>
            <users-count> 0 </users-count>
            <roles-count> 0 </roles-count>
            <indirect-rules-count> 0 </indirect-rules-count>
        </count>
    </T>
endmodule
```

4 Semantics part

This part is for the semantics of the SELinux policy language, expressed as rewrite rules in $\mathbb{K}$ framework, including the core statements which are directly related to the access space in the SELinux policy

language. Some rules are commented to reduce compilation time for special policies where related rules will not be matched.

4.1 Object class statements

```
module CLASS
    imports BOOL
    imports SET
    imports CONFIGURATION

    // class declaration
    rule <k> class C:ClassId => . ...</k>
        <classes> S:Set => S SetItem(C) </classes>
        <class-perms> M:Map => M [C <- .Set] </class-perms>
        requires notBool C in S
    // requires the class undeclared


    // associate a class with a permission list
    rule <k> class C:ClassId { PS:PermIds } => . ...</k>
        <classes> S:Set </classes>
        <class-perms> M:Map => M [C <- PermList2Set(PS)] </class-perms>
        requires C in S
    // requires the class already declared

    // associate a class with a common
    rule <k> class C:ClassId inherits CM:CommonId => . ...</k>
        <common-perms>... CM |-> PERMS:Set ...</common-perms>
        <classes> CLASSES:Set </classes>
        <commons> COMMONS:Set </commons>
        <class-perms>... C |-> (S:Set => S |Set PERMS) ...</class-perms>
        requires C in CLASSES andBool CM in COMMONS
    // requires the class and the common already declared

    // associate a class with a common and permissons
    rule <k> class C:ClassId inherits CM:CommonId { PS:PermIds } => . ...</k>
        <common-perms>... CM |-> PERMS:Set ...</common-perms>
        <classes> CLASSES:Set </classes>
        <class-perms>
            ... C |-> (S:Set => S |Set PERMS |Set PermList2Set(PS)) ...
        </class-perms>
        <commons> COMMONS </commons>
        requires C in CLASSES andBool CM in COMMONS
    // requires the class and the common already declared


    // common permission statement
    rule <k> common C:CommonId { PS:PermIds } => . ...</k>
        <commons> COMMONS:Set => COMMONS |Set SetItem(C) </commons>
        <common-perms> M:Map => M [ C <- PermList2Set(PS) ] </common-perms>
        requires (notBool C in COMMONS) andBool (notBool C in_keys(M))
    // reuqires the common undeclared

    syntax Set ::= PermList2Set(PermIds) [function]
    rule PermList2Set(.PermIds) => .Set
    rule PermList2Set(P:PermId PS:PermIds) => SetItem(P) |Set PermList2Set(PS)

endmodule
```

4.2 Type statements

```
module TYPE
    imports BOOL
    imports SET
    imports MAP
    imports CONFIGURATION

    // attribute declaration
    rule <k> (attribute A:AttrId ;):AttrDeclStmt => . ...</k>
        <attrs> S:Set => S |Set SetItem(A) </attrs>
        <types> TYPES:Set </types>
        <type-alias> ALIASES:Map </type-alias>
        <attr-types> M:Map => M [A <- .Set] </attr-types>
        requires notBool A in S
            andBool notBool A in TYPES
            andBool notBool A in_keys(ALIASES)

    // type declaration
    rule <k> (type T:TypeId ;):TypeDeclStmt => . ...</k>
```

```
    <types> S:Set => S |Set SetItem(T) </types>
    <attrs> ATTRS:Set </attrs>
    <type-alias> ALIAS:Map </type-alias>
    requires notBool T in S
        andBool notBool T in ATTRS
        andBool notBool T in_keys(ALIAS)
// Attributes are in the same namespace as types and aliases

// type declaration with alias
rule <k> type T:TypeId alias { AS:AliasIds } ;
        => (type T ;):TypeDeclStmt
        ~> typealias T alias { AS } ;
    ...</k>

rule <k> type T:TypeId alias A:AliasId ; => . ...</k>
    <types> TYPES:Set => TYPES |Set SetItem(T) </types>
    <attrs> ATTRS:Set </attrs>
    <type-alias> M:Map => M [ A <- T ] </type-alias>
    requires notBool T in TYPES
        andBool notBool T in ATTRS
        andBool notBool T in_keys(M)
        andBool notBool A in_keys(M)


// type declaration with attribute
rule <k> type T:TypeId , ( A:AttrId , AS:AttrList => AS ) ; ... </k>
    <types> TYPES:Set </types>
    <attrs> ATTRS:Set </attrs>
    <attr-types>
        M:Map => M [ A <- SetItem(T) |Set {M[A] orDefault .Set}:>Set ]
    </attr-types>
    requires (notBool T in TYPES) andBool (A in ATTRS)

rule <k> type T:TypeId , .AttrList ; => . ...</k>
    <types> S:Set => S |Set SetItem(T) </types>
    <attrs> ATTRS:Set </attrs>
    requires (notBool T in S) andBool (notBool T in ATTRS)


// type declaration with both attribute and alias
rule <k> type T:TypeId alias ALIASES:Aliases , ATTRS:AttrList ;
        => (type T ;):TypeDeclStmt
        ~> typealias T alias ALIASES ;
        ~> typeattribute T ATTRS ;
    ...</k>
// simply split


// type alias statement
rule <k> typealias T:TypeId alias { A:AliasId AL:AliasIds } ;
        => typealias T alias A ;
        ~> typealias T alias { AL } ;
    ...</k>

rule <k> typealias T:TypeId alias { .AliasIds } ; => . ...</k>
    <types> TYPES:Set </types>
    requires T in TYPES

rule <k> typealias T:TypeId alias A:AliasId ; => . ...</k>
    <type-alias> M:Map => M [ A <- T ] </type-alias>
    <types> TYPES:Set </types>
    requires T in TYPES andBool notBool A in_keys(M)


// type attribute statement
// implicitly requires attribute defined
rule <k> typeattribute T:TypeId ( A:AttrId , AL:AttrList => AL ) ; ... </k>
    <types> TYPES:Set </types>
    <attr-types>
        M:Map => M [ A <- SetItem(T) |Set {M[A] orDefault .Set}:>Set ]
    </attr-types>
    requires T in TYPES

rule <k> typeattribute _T:TypeId .AttrList ; => . ...</k>


// type transition
// rule <k> type_transition _:Types _:Types : _:Classes _:TypeId ; => . ... </k>
// rule <k> type_transition _:Types _:Types : _:Classes _:TypeId _:String ; => . ... </k>
// rule <k> type_change _:Types _:Types : _:Classes _:TypeId ; => . ... </k>
// rule <k> type_member _:Types _:Types : _:Classes _:TypeId ; => . ... </k>
rule <k> _:TypeTransition => . ... </k>
rule <k> _:TypeChangeStmt => . ... </k>
rule <k> _:TypeMemberStmt => . ... </k>
```

```
    syntax Set ::= AttrList2Set(AttrList) [function]
    rule AttrList2Set(.AttrList) => .Set
    rule AttrList2Set(A:AttrId , AS:AttrList) => SetItem(A) |Set AttrList2Set(AS)

endmodule
```

4.3 Modular policy Support

```
module REQUIRE
    imports CONFIGURATION
    imports K-EQUAL

    // rule <k> _RS:RequireStmt => . ...</k>
    syntax Controller ::= "requireFail"

    rule <k> require { .RequiredList } => . ...</k>
    rule <k> require { RS:RequiredStmt RL:RequiredList }
           => require RS ~> require { RL } ...</k>

    rule <k> requireFail ~> R:RequireStmt => . ...</k>
         <require-failed> S:Set => S SetItem(R) </require-failed>

    rule <k> require class C:ClassId ;
           => #if notBool C in S
              #then requireFail ~> require class C ;
              #else . #fi
           ...
         </k>
         <classes> S:Set </classes>


    rule <k> require class C:ClassId P:PermId ;
           => #if notBool P in {M[C] orDefault .Set}:>Set
              #then requireFail ~> require class C P ;
              #else . #fi
           ...
         </k>
         <class-perms> M:Map </class-perms>

    rule <k> require class _:ClassId { .PermIds } ; => . ...</k>
    rule <k> require class C:ClassId { P:PermId PS:PermIds } ;
           => #if notBool P in {M[C] orDefault .Set}:>Set
              #then requireFail ~> require class C P ;
              #else . #fi
           ~> require class C { PS } ;
           ...
         </k>
         <class-perms> M:Map </class-perms>

    rule <k> require role .RoleList ; => . ...</k>
    rule <k> require role R:RoleId , RL:RoleList ;
           => #if notBool R in S
              #then requireFail ~> require role R ;
              #else . #fi
           ~> require role RL ;
           ...
         </k>
         <roles> S:Set => S SetItem(R) </roles>

    rule <k> require type .TypeList ; => . ...</k>
    rule <k> require type T:TypeId , TL:TypeList ;
           => #if notBool T in S
              #then requireFail ~> require type T ;
              #else . #fi
           ~> require type TL ;
           ...
         </k>
         <types> S:Set => S SetItem(T) </types>

    rule <k> require attribute .AttrList ; => . ...</k>
    rule <k> require attribute A:AttrId , AL:AttrList ;
           => #if notBool A in S
              #then requireFail ~> require attribute A ;
              #else . #fi
           ~> require attribute AL ;
           ...
         </k>
         <attrs> S:Set => S SetItem(A) </attrs>

    rule <k> require bool .BoolList ; => . ... </k>
```

```
    rule <k> require bool B:BoolId, BL:BoolList ;
           => #if notBool B in_keys(M)
              #then requireFail ~> require bool B ;
              #else . #fi
           ~> require bool BL ;
           ...
         </k>
         <bools> M:Map => M [ B <- M[B] orDefault false ] </bools>

    rule <k> require attribute_role A:AttrId ;
           => #if notBool A in_keys(M)
              #then requireFail ~> require attribute_role A ;
              #else . #fi
           ...
         </k>
         <attr-roles> M:Map => M [ A <- M[A] orDefault .Set ] </attr-roles>
endmodule
```

4.4 Access vector rule statements

```
module ACCESS
    imports CONFIGURATION
    imports SET
    imports BOOL
    imports MAP
    imports INT
    imports K-EQUAL
    // imports TYPE-UTILS

    syntax Controller ::= "rmDash" | "noDash" | "noSelf"

    rule <k> R:AccessRule T1:TypeItemSet T2:TypeItemSet : { C:ClassId Cs:ClassItems } P:
        PermItemSet ;
           => R T1 T2 : C P ;
           ~> R T1 T2 : { Cs } P ;
           ...
         </k>

    rule <k> _:AccessRule _:TypeItemSet _:TypeItemSet : { .ClassItems } _:PermItemSet ; => . ...
         </k>

    rule <k> R:AccessRule T1:TypeItemSet T2:TypeItemSet : C:ClassId { Ps:PermItems } ;
           => rmDash ~> R T1 T2 : C {Ps} ;
           ...
         </k>

    rule <k> R:AccessRule T1:TypeItemSet T2:TypeItemSet : C:ClassId P:PermId ;
           => rmDash ~> R T1 T2 : C P ;
           ...
         </k>

    rule <k> R:AccessRule T1:TypeItemSet T2:TypeItemSet : C:ClassId * ;
           => rmDash
           ~> R T1 T2 : C { permItemsFromSet({M[C] orDefault .Set}:>Set) } ;
           ...
         </k>
         <class-perms> M:Map </class-perms>

    rule <k> R:AccessRule T1:TypeItemSet T2:TypeItemSet : C:ClassId ~ { Ps:PermItems } ;
           => rmDash
           ~> R T1 T2 : C { permItemsFromSet( {M[C] orDefault .Set}:>Set -Set permItemSet2Set({
               Ps}) ) } ;
           ...
         </k>
         <class-perms> M:Map </class-perms>

    rule <k> R:AccessRule T1:TypeItemSet T2:TypeItemSet : C:ClassId ~ P:PermId ;
           => rmDash
           ~> R T1 T2 : C { permItemsFromSet({M[C] orDefault .Set}:>Set -Set SetItem(P)) } ;
           ...
         </k>
         <class-perms> M:Map </class-perms>



    rule <k> rmDash
           ~> R:AccessRule T1:TypeItemSet T2:TypeItemSet : C:ClassId P:PermItemSet ;
           => noDash
           ~> R rmDashType(T1) rmDashType(T2) : C rmDashPerm(P, C) ;
           ...
         </k>
```

```
// ------------------- process type ----------------------
syntax TypeItemSet ::= "rmDashType" "(" TypeItemSet ")" [function]
rule rmDashType(*) => *
rule rmDashType(TS:TypeItemSet) => { typeItemsFromSet(resolveDashType(TS)) } [owise]

syntax TypeItems ::= "typeItemsFromSet" "(" Set ")" [function]
rule typeItemsFromSet(SetItem(T:TypeId) S:Set) => T typeItemsFromSet(S)
rule typeItemsFromSet(.Set) => .TypeItems


syntax Set ::= "resolveDashType" "(" TypeItemSet ")" [function]
rule [[ resolveDashType(~{ TS:TypeItems }) => TYPES -Set resolveDashType({ TS }) ]]
    <types> TYPES:Set </types>
rule [[ resolveDashType(~T:TypeId) => TYPES -Set resolveTypeAttr(T) ]]
    <types> TYPES:Set </types>
rule resolveDashType({ TS:TypeItems }) => getTypeInclusion(TS) -Set getTypeExclusion(TS)
rule resolveDashType(T:TypeId) => resolveTypeAttr(T)

// rule resolveDashType(_, _M:Map, _S:Set) => .Set [owise]


syntax Set ::= "resolveTypeAttr" "(" TypeId ")" [function]
rule [[ resolveTypeAttr(A:TypeId) => {M[A] orDefault SetItem(A)}:>Set ]]
    <attr-types> M:Map </attr-types>
rule resolveTypeAttr(_) => .Set [owise]


syntax Set ::= "getTypeInclusion" "(" TypeItems ")" [function]
rule getTypeInclusion(T:TypeId Ts:TypeItems)   => resolveTypeAttr(T) |Set getTypeInclusion(
    Ts)
rule getTypeInclusion(- _:TypeId Ts:TypeItems) => getTypeInclusion(Ts)
rule getTypeInclusion(.TypeItems) => .Set
rule getTypeInclusion(_) => .Set [owise]


syntax Set ::= "getTypeExclusion" "(" TypeItems ")" [function]
rule getTypeExclusion(_:TypeId Ts:TypeItems)   => getTypeExclusion(Ts)
rule getTypeExclusion(- T:TypeId Ts:TypeItems) => resolveTypeAttr(T) |Set getTypeExclusion(
    Ts)
rule getTypeExclusion(.TypeItems) => .Set
rule getTypeExclusion(_) => .Set [owise]


// -------------- process class -----------------
syntax ClassItemSet ::= "rmDashClass" "(" ClassItemSet ")" [function]
rule rmDashClass(*) => *
rule rmDashClass(CS:ClassItemSet) => CS [owise]


// -------------- process permission -----------------
syntax PermItemSet ::= "rmDashPerm" "(" PermItemSet "," ClassId ")" [function]
rule rmDashPerm(*, _:ClassId) => *
rule rmDashPerm(Ps:PermItemSet, C:ClassId)
    => { permItemsFromSet(resolveDashPerm(Ps, C)) } [owise]

syntax Set ::= "resolveDashPerm" "(" PermItemSet "," ClassId ")" [function]
rule [[ resolveDashPerm(~{ Ps:PermItems }, C:ClassId)
        => {M[C] orDefault .Set}:>Set -Set resolveDashPerm({Ps}, C) ]]
    <class-perms> M:Map </class-perms>
rule resolveDashPerm({ P:PermId Ps:PermItems }, C:ClassId)
    => SetItem(P) |Set resolveDashPerm({Ps}, C)
rule resolveDashPerm({ .PermItems }, _:ClassId) => .Set
rule resolveDashPerm(P:PermId, _:ClassId) => SetItem(P)
rule resolveDashPerm(_, _) => .Set [owise]


// -------------- no dash process ---------------
rule <k> noDash
        ~> R:AccessRule { T1:TypeId TS1:TypeItems } TS2:TypeItemSet : C:ClassId PS:
            PermItemSet ;
        => noDash ~> R T1 TS2 : C PS ;
        ~> noDash ~> R { TS1 } TS2 : C PS ;
        ...
    </k>

rule <k> noDash
        ~> _:AccessRule { .TypeItems } _:TypeItemSet : _:ClassId _:PermItemSet ;
        => . ...
    </k>

rule <k> noDash
        ~> R:AccessRule T1:TypeId { T2:TypeId TS2:TypeItems } : C:ClassId PS:PermItemSet ;
        => noDash ~> R T1 T2 : C PS ;
```

```
        ~> noDash ~> R T1 { TS2 } : C PS ;
        ...
    </k>

rule <k> noDash
        ~> _:AccessRule _:TypeId { .TypeItems } : _:ClassId _:PermItemSet ;
        => . ...
    </k>

// when T1 is *
rule <k> noDash
        ~> R:AccessRule * { T2:TypeId TS2:TypeItems } : C:ClassId PS:PermItemSet ;
        => noDash ~> R * T2 : C PS ;
        ~> noDash ~> R * { TS2 } : C PS ;
        ...
    </k>

rule <k> noDash
        ~> _:AccessRule * { .TypeItems } : _:ClassId _:PermItemSet ;
        => . ...
    </k>


// only read and write
rule <k> noDash
        ~> R:AccessRule T1:TypeId T2:TypeId : C:ClassId { P:PermId PS:PermItems } ;
        => #if true // P ==K #token("read", "Id") orBool P ==K #token("write", "Id")
            #then noDash ~> R T1 T2 : C P ;
            #else . #fi
        ~> noDash ~> R T1 T2 : C { PS } ;
        ...
    </k>

rule <k> noDash
        ~> R:AccessRule T1:TypeId * : C:ClassId { P:PermId PS:PermItems } ;
        => #if true // P ==K #token("read", "Id") orBool P ==K #token("write", "Id")
            #then noDash ~> R T1 * : C P ;
            #else . #fi
        ~> noDash ~> R T1 * : C { PS } ;
        ...
    </k>

rule <k> noDash
        ~> R:AccessRule * T2:TypeId : C:ClassId { P:PermId PS:PermItems } ;
        => #if true // P ==K #token("read", "Id") orBool P ==K #token("write", "Id")
            #then noDash ~> R * T2 : C P ;
            #else . #fi
        ~> noDash ~> R * T2 : C { PS } ;
        ...
    </k>

rule <k> noDash
        ~> R:AccessRule * * : C:ClassId { P:PermId PS:PermItems } ;
        => #if true // P ==K #token("read", "Id") orBool P ==K #token("write", "Id")
            #then noDash ~> R * * : C P ;
            #else . #fi
        ~> noDash ~> R * * : C { PS } ;
        ...
    </k>

rule <k> noDash ~> _:AccessRule _:TypeId _:TypeId : _:ClassId { .PermItems } ; => . ...</k>
rule <k> noDash ~> _:AccessRule _:TypeId     *    : _:ClassId { .PermItems } ; => . ...</k>
rule <k> noDash ~> _:AccessRule     *    _:TypeId : _:ClassId { .PermItems } ; => . ...</k>
rule <k> noDash ~> _:AccessRule     *        *    : _:ClassId { .PermItems } ; => . ...</k>

// substitute self
rule <k> noDash
        ~> R:AccessRule T1:TypeId T2:TypeId : C:ClassId P:PermId ;
        => noSelf
        ~> #if T2 ==K #token("self", "Id")
            #then R T1 T1 : C P ;
            #else R T1 T2 : C P ;
            #fi
        ...
    </k>

rule <k> noDash ~> R:AccessRule T1:TypeId * : C:ClassId P:PermId ;
        => noSelf ~> R T1 * : C P ;
        ...
    </k>

rule <k> noDash ~> R:AccessRule * T2:TypeId : C:ClassId P:PermId ;
        => noSelf ~> R * T2 : C P ;
        ...
```

```
    </k>

rule <k> noDash ~> R:AccessRule * * : C:ClassId P:PermId ;
        => noSelf ~> R * * : C P ;
        ...
    </k>


// allow
rule <k> noSelf
        ~> allow T1:TypeId T2:TypeId : C:ClassId P:PermId ;
        => . ...
    </k>
    <allow> L:Set => L |Set SetItem(allow T1 T2 : C P) </allow>
    <domains> S1:Set => S1 |Set SetItem(T1) </domains>
    <objects> S2:Set => S2 |Set SetItem(T2 : C) </objects>

// neverallow
// rule <k> noSelf ~> neverallow T1:TypeName T2:TypeName : C:ClassName P:PermName ; => . ...
     </k>
//     <neverallow> L:Set => L SetItem(neverallow T1 T2 : C P) </neverallow>
//     <domains> S1:Set => S1 SetItem(T1) </domains>
//     <objects> S2:Set => S2 SetItem(T2 : C) </objects>

rule <k> noSelf ~> neverallow T1:TypeId T2:TypeId : C:ClassId P:PermId ; => . ... </k>
    <neverallow-map>
        M:Map => M [ (T1 T2 C) <- SetItem(P) |Set {M[(T1 T2 C)] orDefault .Set}:>Set ]
    </neverallow-map>
    <neverallow> S:Set => S |Set SetItem(neverallow T1 T2 : C P) </neverallow>

rule <k> noSelf ~> neverallow T1:TypeId * : C:ClassId P:PermId ; => . ... </k>
    <neverallow-map>
        M:Map => M [ (T1 * C) <- SetItem(P) |Set {M[(T1 * C)] orDefault .Set}:>Set ]
    </neverallow-map>
    <neverallow> S:Set => S |Set SetItem(neverallow T1 * : C P) </neverallow>

rule <k> noSelf ~> neverallow * T2:TypeId : C:ClassId P:PermId ; => . ... </k>
    <neverallow-map>
        M:Map => M [ (* T2 C) <- SetItem(P) |Set {M[(* T2 C)] orDefault .Set}:>Set ]
    </neverallow-map>
    <neverallow> S:Set => S |Set SetItem(neverallow * T2 : C P) </neverallow>

rule <k> noSelf ~> neverallow * * : C:ClassId P:PermId ; => . ... </k>
    <neverallow-map>
        M:Map => M [ (* * C) <- SetItem(P) |Set {M[(* * C)] orDefault .Set}:>Set ]
    </neverallow-map>
    <neverallow> S:Set => S |Set SetItem(neverallow * * : C P) </neverallow>

// dontaudit
rule <k> noSelf ~> dontaudit _:TypeId _:TypeId : _:ClassId _:PermId ; => . ... </k>

// auditallow
rule <k> noSelf ~> auditallow _:TypeId _:TypeId : _:ClassId _:PermId ; => . ... </k>

// auditdeny
rule <k> noSelf ~> auditdeny _:TypeId _:TypeId : _:ClassId _:PermId ; => . ... </k>


// ----------------- record allow and neverallow -----------------

// allow
// rule <k> noSelf ~> allow T1:TypeName T2:TypeName : C:ClassName PS:PermItemSet ; => .
     ...</k>
//     <allow>
//         M:Map => M [ (T1 T2 C) <- { permItemsFromSet(
//             permItemSet2Set({M[(T1 T2 C)] orDefault {.PermItems}}:>PermItemSet) |Set
     permItemSet2Set(PS)
//         ) } ]
//     </allow>

// neverallow
// rule <k> noSelf ~> neverallow T1:TypeName T2:TypeName : C:ClassName PS:PermItemSet ; => .
      ...</k>
//     <neverallow>
//         M:Map => M [ (T1 T2 C) <- { permItemsFromSet(
//             permItemSet2Set({M[(T1 T2 C)] orDefault {.PermItems}}:>PermItemSet) |Set
     permItemSet2Set(PS)
//         ) } ]
//     </neverallow>

// rule <k> noSelf ~> neverallow T1:TypeName * : C:ClassName PS:PermItemSet ; => . ...</k>
//     <neverallow> M:Map => M [(T1 * C) <- PS] </neverallow>

// rule <k> noSelf ~> neverallow * T2:TypeName : C:ClassName PS:PermItemSet ; => . ...</k>
```

```
    //     <neverallow> M:Map => M [(* T2 C) <- PS] </neverallow>


    // auditallow
    // rule <k> noSelf ~> auditallow _T1:TypeName _T2:TypeName : _C:ClassName _PS:PermItemSet ;
         => . ...</k>
    //     <auditallow> M => M [(T1 T2 C) <- PS] </auditallow>

    // dontaudit
    // rule <k> noSelf ~> dontaudit _T1:TypeName _T2:TypeName : _C:ClassName _PS:PermItemSet ;
         => . ...</k>
    //     <dontaudit> M => M [(T1 T2 C) <- PS] </dontaudit>

    // auditdeny
    // rule <k> noSelf ~> auditdeny _T1:TypeName _T2:TypeName : _C:ClassName _PS:PermItemSet ;
         => . ...</k>
    //     <auditdeny> M => M [(T1 T2 C) <- PS] </auditdeny>



    syntax Set ::= "permItemSet2Set" "(" PermItemSet ")" [function]
    rule permItemSet2Set(P:PermId) => SetItem(P)
    rule permItemSet2Set({P:PermId Ps:PermItems}) => SetItem(P) |Set permItemSet2Set({Ps})
    rule permItemSet2Set({.PermItems}) => .Set
    rule permItemSet2Set(_) => .Set [owise]

    syntax PermItems ::= "permItemsFromSet" "(" Set ")" [function]
    rule permItemsFromSet(SetItem(P:PermId) S:Set) => P permItemsFromSet(S)
    rule permItemsFromSet(.Set) => .PermItems
    rule permItemsFromSet(_) => .PermItems [owise]

endmodule
```

4.5 Multi-level security statements

```
module MLS
    imports CONFIGURATION
    imports SET
    imports BOOL
    imports MAP

    rule <k> (sensitivity SN:SensId ;):SensDecl => . ...</k>
        <sensitivities> S:Set => S SetItem(SN) </sensitivities>
        requires notBool SN in S

    rule <k> sensitivity SN:SensId alias SAN:SensAliasName SANs:SensAliasNames ;
            => sensitivity SN alias SANs ;
            ...
        </k>
        <sens-alias> M:Map => M [ SAN <- SN ] </sens-alias>

    rule <k> sensitivity SN:SensId alias .SensAliasNames ; => . ...</k>
        <sensitivities> S => S:Set SetItem(SN) </sensitivities>
        // requires notBool SN in S

    // todo: may exist some problems
    rule <k> level SN:SensId ; => . ...</k>
        <levels> M:Map => M [ SN <- .CategorySet ] </levels>
        // requires notBool SN in_keys(M)

    rule <k> level SN:SensId : CS:CategorySet ; => . ...</k>
        <levels> M:Map => M [ SN <- CS ] </levels>
        // <categories> CATEGORIES </categories>
        // requires CS in CATEGORIES andBool notBool L in S

    rule <k> (category CN:CategoryId ;):CategoryDecl => . ...</k>
        <categories> S:Set => S SetItem(CN) </categories>
        requires notBool CN in S

    rule <k> category CN:CategoryId alias CAN:CategoryAliasName CANs:CategoryAliasNames ;
            => category CN alias CANs ;
        ...</k>
        <categories> CATEGORIES </categories>
        <category-alias> M:Map => M [ CAN <- CN ] </category-alias>
        requires notBool CN in CATEGORIES andBool notBool CAN in_keys(M)

    rule <k> category CN:CategoryId alias .CategoryAliasNames ; => . ...</k>
        <categories> S:Set => S SetItem(CN) </categories>
        requires notBool CN in S

    rule <k> dominance { SNs:SensIds } => . ...</k>
        <dominances> S:Set => S SetItem(SNs) </dominances>
```

```
    // rule <k> mlsconstrain _CS:ClassSet _PS:PermSet _E:MlsExp ; => . ...</k>
    rule <k> mlsconstrain { C:ClassId CNs:ClassIds } PS:PermSet E:MlsExp ;
            => mlsconstrain C PS E ;
            ~> mlsconstrain { CNs } PS E ;
        ...</k>

    rule <k> mlsconstrain { .ClassIds } _PS:PermSet _E:MlsExp ; => . ...</k>

    rule <k> mlsconstrain C:ClassId { P:PermId PS:PermIds } E:MlsExp ;
            => mlsconstrain C P E ;
            ~> mlsconstrain C { PS } E ;
        ...</k>

    rule <k> mlsconstrain _C:ClassId { .PermIds } _E:MlsExp ; => . ...</k>

    // rule <k> mlsconstrain C:ClassId P:PermId E:MlsExp ; => . ...</k>
    //     <mlsconstrains> M => M (C P) |-> SetItem(E) </mlsconstrains>
    //     requires notBool (C P) in_keys(M)
    // rule <k> mlsconstrain C:ClassId P:PermId E:MlsExp ; => . ...</k>
    //     <mlsconstrains>... (C P) |-> S => (C P) |-> S SetItem(E) ...</mlsconstrains>
    rule <k> mlsconstrain C:ClassId P:PermId E:MlsExp ; => . ...</k>
        <mlsconstrains>
            M:Map => M [(C P) <- SetItem(E) |Set {M[(C P)] orDefault .Set}:>Set]
        </mlsconstrains>
        requires ( isWritePerm(P) impliesBool canMlsWrite(E) )
            andBool ( isReadPerm(P) impliesBool canMlsRead(E) )

    rule <k> mlsvalidatetrans _CS:ClassSet _ME:MlsExp ; => . ...</k>

    syntax ClassPermPair ::= "(" ClassId PermId ")"


    // syntax Bool ::= isMlsSafe(MlsExp) [function]

    // rule isMlsSafe(E1:MlsExp or E2:MlsExp)  => isMlsSafe(E1) orBool  isMlsSafe(E2)
    // rule isMlsSafe(E1:MlsExp and E2:MlsExp) => isMlsSafe(E1) andBool isMlsSafe(E2)
    // rule isMlsSafe(_E1:Name _Op:BoolOp _E2:Name) => false
    // rule isMlsSafe(_) => true [owise]

    syntax Bool ::= canMlsWrite(MlsExp) [function]
    syntax Bool ::= canMlsRead(MlsExp) [function]

    rule canMlsRead(_) => false
    rule canMlsWrite(_) => false
endmodule
```

4.6 User & Role statements

Rewrite rules for user and role statements.

```
module ROLE
    imports CONFIGURATION
    imports SET
    imports BOOL
    imports LIST

    // rule <k> _:RoleDeclStmt => . ...</k>
    rule <k> (role R:RoleId ;):RoleDeclStmt => . ...</k>
        <roles> S:Set => S SetItem(R) </roles>
        // requires notBool R in S

    rule <k> role R:RoleId types T:TypeId ; => . ...</k>
        <roles> S:Set => S SetItem(R) </roles>
        <role-types>
            M:Map => M [ R <- SetItem(T) |Set {M[R] orDefault .Set}:>Set ]
        </role-types>
        // requires notBool R in S


    // role declaration
    // There can be multiple role statements for the same role identifier.
    // rule <k> RD:RoleDecl => mark_RoleDecl RD ...</k>


    // rule <k> mark_RoleDecl role R:RoleId ; => . ...</k>
        // <roles> S => S SetItem(R) </roles>
    // requires notBool R in S

    // rule <k> mark_RoleDecl role R:RoleId types TS:TypeSet ; => . ...</k>
        // <roles> S => S SetItem(R) </roles>
        // <role-types> M => M R |-> TypeSet2Set(TS) </role-types>
```

```
// requires notBool R in S


// role dominance
rule <k> _:RoleDomStmt => . ...</k>
// rule <k> dominance { D:Dominance } => . ...</k>
//     <dominance> S => S SetItem(DominanceList(D)) </dominance>
// rule <k> mark_Dominance _:RoleItems { _:Dominance } => . ...</k>
// rule <k> mark_Dominance _:RoleItems ; => . ...</k>
// todo: dominance detail


// role allow statement
// rule <k> allow { R:RoleId RNs:RoleIds } RS:RoleSet ;
//         => allow R RS ;
//         ~> allow { RNs } RS ;
//     ...</k>
// rule <k> allow { .RoleIds } _RS:RoleSet ; => . ...</k>

rule <k> _:RoleAllowStmt => . ...</k>
// rule <k> allow R:RoleId RS:RoleSet ; => . ...</k>
//     <role-allow> M => M R |-> RoleSet2Set(RS) </role-allow>
// requires notBool R in_keys(M)

// rule <k> allow R:RoleId RS:RoleSet ; => . ...</k>
//     <role-allow>... R |-> (S => S RoleSet2Set(RS)) ...</role-allow>

// rule <k> allow R:RoleId RS:RoleSet ; => . ...</k>
//     <role-allow>
//         M:Map => M [R <- RoleSet2Set(RS) |Set {M[R] orDefault .Set}:>Set]
//     </role-allow>
// requires notBool R in_keys(M)


rule <k> _:RoleTransitionStmt => . ...</k>
// role transition statement
// rule <k> role_transition { R1:RoleId RNs:RoleIds } TS:TypeSet R:RoleId ;
//         => role_transition R1 TS R ;
//         ~> role_transition { RNs } TS R ;
//     ...</k>
// rule <k> role_transition { .RoleIds } _TS:TypeSet _R2:RoleId ; => . ...</k>

// rule <k> role_transition R1:RoleId { T:TypeId TNs:TypeIds } R2:RoleId ;
//         => role_transition R1 T R2 ;
//         ~> role_transition R1 { TNs } R2 ;
//     ...</k>
// rule <k> role_transition _R1:RoleId { .TypeIds } _R2:RoleId ; => . ...</k>

// rule <k> role_transition R1:RoleId T:TypeId R2:RoleId ; => . ...</k>
//     <roleTrans> S => S SetItem((R1 T R2)) </roleTrans>
// todo: ensure types declared


// user declaration
rule <k> user U:UserId roles _R:Roles ; => . ...</k>
    <users> S:Set => S SetItem(U) </users>
    // <user-roles> M:Map => M [U <- RoleSet2Set(R)] </user-roles>

rule <k> user U:UserId roles _:Roles level _:LevelId range _:MlsRange ; => . ... </k>
    <users> S:Set => S SetItem(U) </users>


// attribute role declaration
rule <k> (attribute_role A:AttrId ;):AttrRoleStmt => . ...</k>
    <attr-roles> M:Map => M [A <- .Set] </attr-roles>
    requires notBool A in_keys(M)

rule <k> roleattribute R:RoleId A:AttrId ; => . ...</k>
    <attr-roles>... A |-> (S:Set => S SetItem(R)) ...</attr-roles>
// requires AR is defined implicitly


// syntax Set ::= RoleSet2Set(Roles) [function]
// rule RoleSet2Set(R:RoleId) => SetItem(R)
// rule RoleSet2Set({ .RoleIds }) => .Set
// rule RoleSet2Set({ R:RoleId RS:RoleIds }) => SetItem(R) RoleSet2Set({ RS })

// syntax Set ::= TypeSet2Set(Types) [function]
// rule TypeSet2Set(T:TypeId) => SetItem(T)
// rule TypeSet2Set({ .TypeIds }) => .Set
// rule TypeSet2Set({ T:TypeId TS:TypeIds }) => SetItem(T) TypeSet2Set({ TS })

// syntax RoleTransRuleItem ::= "(" RoleId TypeId RoleId ")"
```

```
    // syntax List ::= DominanceList(Dominance) [function]
    // rule DominanceList(R:RoleItems ;) => ListItem(R)
    // rule DominanceList(R:RoleItems { D:Dominance }) => ListItem(R) DominanceList(D)

endmodule
```

4.7 Constraint statements

```
module CONSTRAINT
    imports CONFIGURATION
    imports SET
    imports MAP
    imports BOOL

    // rule <k> _:ConstrainStmt => . ...</k>

    rule <k> validatetrans { C:ClassId CNs:ClassIds } CE:ConstraintExp ;
           => validatetrans C CE ; ~> validatetrans { CNs } CE ;
        ...</k>
    rule <k> validatetrans { .ClassIds } _CE:ConstraintExp ; => . ...</k>

    rule <k> validatetrans C:ClassId CE:ConstraintExp ; => . ...</k>
        <validatetrans>
            M:Map => M [ C <- SetItem(CE) |Set {M[C] orDefault .Set}:>Set ]
        </validatetrans>

    // constrain statement
    rule <k> constrain { C:ClassId CNs:ClassIds } PS:Perms ( CE:ConstraintExp ) ;
           => constrain C PS ( CE ) ; ~> constrain { CNs } PS ( CE ) ;
        ...</k>
    rule <k> constrain { .ClassIds } _PS:Perms ( _CE:ConstraintExp ) ; => . ...</k>

    rule <k> constrain C:ClassId { P:PermId PS:PermIds } ( CE:ConstraintExp ) ;
           => constrain C P ( CE ) ; ~> constrain C { PS } ( CE ) ;
        ...</k>
    rule <k> constrain _C:ClassId { .PermIds } ( _CE:ConstraintExp ) ; => . ...</k>

    rule <k> constrain C:ClassId P:PermId ( CE:ConstraintExp ) ; => . ...</k>
        <constraints>
            M:Map => M [ (C P) <- SetItem(CE) |Set {M[(C P)] orDefault .Set}:>Set ]
        </constraints>
endmodule
```

4.8 Condition statements

```
module CONDITION
    imports CONFIGURATION
    imports BOOL
    imports MAP
    imports K-EQUAL

    rule <k> bool BN:BoolId B:Bool ; => . ...</k>
        <bools> M:Map => M [ BN <- B ] </bools>

    rule <k> if ( CE:CondExp ) { SL:StmtList }
           => if ( CE ) { SL } else { .StmtList }
            ...
        </k>

    rule <k> if ( CE:CondExp ) { SL1:StmtList } else { SL2:StmtList }
           => #if CondExpEval(CE) #then { SL1 } #else { SL2 } #fi
            ...
        </k>

    rule <k> { S:Stmt SL:StmtList } => S ~> { SL } ... </k>
    rule <k> { .StmtList } => . ... </k>

    syntax Bool ::= "CondExpEval" "(" CondExp ")" [function]
    rule CondExpEval(! E:CondExp) => notBool CondExpEval(E)
    rule CondExpEval(E1:CondExp && E2:CondExp) => CondExpEval(E1) andBool CondExpEval(E2)
    rule CondExpEval(E1:CondExp || E2:CondExp) => CondExpEval(E1) orBool  CondExpEval(E2)
    rule CondExpEval(E1:CondExp != E2:CondExp) => CondExpEval(E1) =/=Bool CondExpEval(E2)
    rule CondExpEval(E1:CondExp == E2:CondExp) => CondExpEval(E1) ==Bool  CondExpEval(E2)
    rule [[ CondExpEval(E:BoolId) => B ]]
        <bools>... E |-> B:Bool ...</bools>
endmodule
```

5 Semantic execution part

This part is for some basic check for policy properties like consistency, and completeness.

5.1 Execution entry

```
module MAIN-SYNTAX
    imports CLASS
    imports TYPE
    imports OTHERS
    imports CONDITION
    imports ROLE
    imports ACCESS
    imports REQUIRE
    imports CONSTRAINT
    imports MLS
    imports CHECK

    rule <k> S:Stmt P:Pgm => S ~> P ... </k>

    rule <k> all-last => . ... </k>
        <allow> As:Set => .Set </allow>
        <neverallow> Ns:Set => .Set </neverallow>
        <neverallow-map> _ => .Map </neverallow-map>
        <indirect-rules-result> IRs => .Set </indirect-rules-result>
        <domains> Ds => .Set </domains>
        <objects> Os => .Set </objects>
        <allow-count>  _ => size(As) </allow-count>
        <neverallow-count> _ => size(Ns) </neverallow-count>
        <domains-count> _ => size(Ds) </domains-count>
        <objects-count> _ => size(Os) </objects-count>
        <indirect-rules-count> _ => size(IRs) </indirect-rules-count>
        <class-perms>   _ => .Map </class-perms>

    rule <k> .Pgm => . ... </k>
        <classes>       C => .Set </classes>
        <common-perms>  _ => .Map </common-perms>
        // <class-perms>   _ => .Map </class-perms>
        <commons>       _ => .Set </commons>
        <types>         T => .Set </types>
        <attrs>         _ => .Set </attrs>
        <type-alias>    _ => .Map </type-alias>
        <attr-types>    _ => .Map </attr-types>
        <bools>         B => .Map </bools>
        <roles>         R => .Set </roles>
        <users>         U => .Set </users>
        <user-roles>    _ => .Map </user-roles>
        <role-types>    _ => .Map </role-types>
        <role-allow>    _ => .Map </role-allow>
        <role-trans>    _ => .Set </role-trans>
        <dominance>     _ => .Set </dominance>
        <sids>          _ => .Map </sids>
        <caps>          _ => .Set </caps>
        <categories>    _ => .Set </categories>
        <bools-count>   _ => size(keys(B)) </bools-count>
        <classes-count> _ => size(C) </classes-count>
        <types-count>   _ => size(T) </types-count>
        <users-count>   _ => size(U) </users-count>
        <roles-count>   _ => size(R) </roles-count>
endmodule

module MAIN
    imports MAIN-SYNTAX
endmodule
```

5.2 Properties checking

```
module CHECK
    imports CONFIGURATION
    imports CHECK-CONSTRAINT
    imports CHECK-COMPLETENESS
    imports CHECK-1
    imports CHECK-2

    // syntax Controller ::= "check"

    rule <k> check
           => check-completeness
           // ~> check-constraint
```

```
            // ~> check1
            // ~> check2
            ...
        </k>
endmodule

module CHECK-CONSTRAINT
    imports CONFIGURATION
    imports K-EQUAL

    syntax Controller ::= "check-constraint"
                        | "constraint-fail"
                        | "constraint-check"
                        | "constraint-check-end"

    rule <k> check-constraint
            => constraint-check ~> Rs
            ~> constraint-check-end
            ...
        </k>
        <allow> Rs:Set </allow>

    rule <k> constraint-check-end => check ...</k>

    rule <k> constraint-fail
            ~> allow T1:TypeId T2:TypeId : C:ClassId P:PermId
            => . ...
        </k>
        <constraint-failed>
            S:Set => S SetItem(allow T1 T2 : C P)
        </constraint-failed>

    rule <k> constraint-check ~> .Set => . ...</k>
    rule <k> constraint-check
            ~> SetItem(allow T1:TypeId T2:TypeId : C:ClassId P:PermId) Rs:Set
            => #if constraintExpsEval(T1 T2 {M[(C P)] orDefault .Set}:>Set)
                #then .
                #else constraint-fail ~> allow T1 T2 : C P
                #fi
            ~> constraint-check ~> Rs
            ...
        </k>
        <constraints> M:Map </constraints>


    syntax Bool ::= "constraintExpsEval" "(" TypeId TypeId Set ")" [function]

    rule constraintExpsEval(_:TypeId _:TypeId .Set) => true

    rule constraintExpsEval(T1:TypeId T2:TypeId SetItem(E:ConstraintExp) Exps:Set)
        => constraintExpEval(T1 T2 E) andBool constraintExpsEval(T1 T2 Exps)

    syntax Bool ::= "constraintExpEval" "(" TypeId TypeId ConstraintExp ")" [function]

    rule constraintExpEval(T1:TypeId T2:TypeId E1:ConstraintExp and E2:ConstraintExp)
        => constraintExpEval(T1 T2 E1) andBool constraintExpEval(T1 T2 E2)

    rule constraintExpEval(T1:TypeId T2:TypeId E1:ConstraintExp or E2:ConstraintExp)
        => constraintExpEval(T1 T2 E1) orBool constraintExpEval(T1 T2 E2)

    // rule constraintExpEval(T1:TypeName T2:TypeName (E:ConstraintExp) )
    //     => constraintExpEval(T1 T2 E)

    rule constraintExpEval(T1:TypeId T2:TypeId A:Name Op:BoolOp B:Name)
        => #if A ==K #token("t1", "Id") orBool A ==K #token("t2", "Id")
            orBool B ==K #token("t1", "Id") orBool B ==K #token("t2", "Id")
        #then
            evalCooledExp(
                {
                    #if A ==K #token("t1", "Id") #then T1
                    #else
                        #if A ==K #token("t2", "Id") #then T2
                        #else A #fi
                    #fi
                }:>Id
                Op
                {
                    #if B ==K #token("t1", "Id") #then T1
                    #else
                        #if B ==K #token("t2", "Id") #then T2
                        #else B #fi
                    #fi
                }:>Id
            )
```

```
        #else
            true
        #fi

    syntax Bool ::= "evalCooledExp" "(" Id BoolOp Id ")" [function]
    rule evalCooledExp(A:Id == B:Id) => A ==K  B
    rule evalCooledExp(A:Id != B:Id) => A =/=K B

endmodule


module CHECK-COMPLETENESS
    imports CONFIGURATION

    syntax Controller ::= "check-completeness"
                        | "completeness-get-rules"
                        | "completeness-get-used-perms-1"
                        | "completeness-get-used-perms-2"
                        | "completeness-get-unused-perms"

    rule <k> check-completeness => completeness-get-rules ... </k>

    rule <k> completeness-get-rules
            => completeness-get-used-perms-1 ~> S1
            ~> completeness-get-used-perms-2 ~> S2
        ... </k>
        <allow> S1:Set </allow>
        <neverallow> S2:Set </neverallow>

    rule <k> completeness-get-used-perms-1
            ~> (SetItem(allow _:TypeId _:TypeId : C:ClassId P:PermId) S:Set => S)
            ...
        </k>
        <perms-used>
            M:Map => M [ C <- SetItem(P) |Set {M[C] orDefault .Set}:>Set ]
        </perms-used>

    rule <k> completeness-get-used-perms-1 ~> .Set => . ... </k>

    rule <k> completeness-get-used-perms-2
            ~> (SetItem(neverallow _:TypeItemSet _:TypeItemSet : C:ClassId P:PermId) S:Set => S)
            ...
        </k>
        <perms-used>
            M:Map => M [ C <- SetItem(P) |Set {M[C] orDefault .Set}:>Set ]
        </perms-used>

    rule <k> completeness-get-used-perms-2 ~> .Set
            => completeness-get-unused-perms ~> keys(M)
            ...
        </k>
        <perms-used> M:Map </perms-used>
        <perms-unused> _ => CP </perms-unused>
        <class-perms> CP:Map </class-perms>


    rule <k> completeness-get-unused-perms ~> (SetItem(C:ClassId) S:Set => S) ... </k>
        <perms-used> M1:Map </perms-used>
        <perms-unused>
            M2:Map => M2 [ C <- {M2[C] orDefault .Set}:>Set -Set {M1[C] orDefault .Set}:>Set ]
        </perms-unused>

    rule <k> completeness-get-unused-perms ~> .Set => . ... </k>
        <perms-used> _ => .Map </perms-used>
        <class-perms> _ => .Map </class-perms>

endmodule


module LABEL-BUILD
    imports LIST
    imports COLLECTIONS
    imports K-EQUAL
    imports CONFIGURATION

    syntax Controller ::= "build-labels"
                        | "build-labels-again"
                        | "get-rules"
                        | "get-rules-1"
                        | "getAllObjects"
                        | "getAllObjects-1"
                        | "getAllObjects-2"
                        | "getAllDomains"
                        | "getAllDomains-1"
```

```
                    | "getAllDomains-2"
                    | "ExtractObjectTypeLabel"
                    | "ExtractObjectTypeLabel-1"
                    | "ExtractObjectTypeLabel-2"
                    | "ExtractDomainLabel"
                    | "ExtractDomainLabel-1"
                    | "ExtractDomainLabel-2"
                    | "ExtractDomainLabel-read"
                    | "ExtractDomainLabel-write"

rule <k> build-labels
        => get-rules
        ~> getAllObjects
        ~> getAllDomains
        ~> ExtractObjectTypeLabel
        ~> ExtractDomainLabel
        ...
    </k>

rule <k> build-labels-again
        => getAllObjects
        ~> getAllDomains
        ~> ExtractObjectTypeLabel
        ~> ExtractDomainLabel
        ...
    </k>

rule <k> get-rules => get-rules-1 ~> Allows ...</k>
    <allow> Allows:Set </allow>

rule <k> get-rules-1 ~> .Set => . ...</k>
rule <k> get-rules-1
        ~> ( SetItem(allow T1:TypeId T2:TypeId : C:ClassId P:PermId)
            Allows:Set => Allows )
        ...
    </k>
    <rules>
        L:List => L {
            #if P ==K #token("read","Id") orBool P ==K #token("write","Id")
            #then ListItem(allow T1 T2 : C P)
            #else .List #fi
        }:>List
    </rules>

// get all objects
rule <k> getAllObjects => getAllObjects-1 ~> 0 ... </k>
    <domains> _ => .Set </domains>
    <objects> _ => .Set </objects>

rule <k> getAllObjects-1 ~> Index:Int
        => #if Index <Int size(L)
            #then getAllObjects-2 ~> L[Index]
                ~> getAllObjects-1 ~> Index +Int 1
            #else . #fi
        ...
    </k>
    <rules> L:List </rules>

rule <k> getAllObjects-2
        ~> allow _:TypeId T2:TypeId : C:ClassId _:PermId
        => . ...
    </k>
    <objects> S:Set => S SetItem(T2 : C) </objects>


// get all domains
rule <k> getAllDomains => getAllDomains-1 ~> 0 ... </k>
rule <k> getAllDomains-1 ~> Index:Int
        => #if Index <Int size(L)
            #then getAllDomains-2 ~> L[Index]
                ~> getAllDomains-1 ~> Index +Int 1
            #else . #fi
        ...
    </k>
    <rules> L:List </rules>

rule <k> getAllDomains-2
        ~> allow T1:TypeId _:TypeId : _:ClassId _:PermId
        => . ...
    </k>
    <domains> S:Set => S SetItem(T1) </domains>


// extract object type label
```

```
rule <k> ExtractObjectTypeLabel
        => ExtractObjectTypeLabel-1 ~> Set2List(S)
        ...
    </k>
    <objects> S:Set </objects>

rule <k> ExtractObjectTypeLabel-1 ~> ( ListItem(O:Object) L:List => L )
    ...</k>
    <object-type-labels>
        <O-write> M1:Map => M1 [ O <- .Set ] </O-write>
        <O-read>  M2:Map => M2 [ O <- .Set ] </O-read>
    </object-type-labels>

rule <k> ExtractObjectTypeLabel-1 ~> .List
        => ExtractObjectTypeLabel-2 ~> L
        ...
    </k>
    <rules> L:List </rules>

rule <k> ExtractObjectTypeLabel-2
        ~> ( ListItem(allow T1:TypeId T2:TypeId : C:ClassId #token("read","Id"))
            Rs:List => Rs )
        ...
    </k>
    <O-read>... T2 : C |-> (S:Set => S SetItem(T1)) ...</O-read>

rule <k> ExtractObjectTypeLabel-2
        ~> ( ListItem(allow T1:TypeId T2:TypeId : C:ClassId #token("write","Id"))
            Rs:List => Rs )
        ...
    </k>
    <O-write>... T2 : C |-> (S:Set => S SetItem(T1)) ...</O-write>

rule <k> ExtractObjectTypeLabel-2
        ~> ( ListItem(neverallow _T1:TypeId _T2:TypeId : _C:ClassId _P:PermId)
            Rs:List => Rs )
        ...
    </k>

rule <k> ExtractObjectTypeLabel-2 ~> .List => . ...</k>


// extract domain labels

rule <k> ExtractDomainLabel => ExtractDomainLabel-1 ~> Set2List(S) ...</k>
    <domains> S:Set </domains>

rule <k> ExtractDomainLabel-1 ~> ( ListItem(D:TypeId) Ds:List => Ds ) ...</k>
    <domain-labels>
        <D-read>  M1:Map => M2 [ D <- S ] </D-read>
        <D-write> M2:Map => M1 [ D <- S ] </D-write>
    </domain-labels>
    <domains> S:Set </domains>

rule <k> ExtractDomainLabel-1 ~> .List
        => ExtractDomainLabel-2 ~> Set2List(S)
        ...
    </k>
    <objects> S:Set </objects>

rule <k> ExtractDomainLabel-2 ~> ListItem(T:Object) Ts:List
        => ExtractDomainLabel-read ~> T ~> Set2List(Rt)
        ~> ExtractDomainLabel-write ~> T ~> Set2List(Wt)
        ~> ExtractDomainLabel-2 ~> Ts
        ...
    </k>
    <O-read>...  T |-> Rt:Set ...</O-read>
    <O-write>... T |-> Wt:Set ...</O-write>

rule <k> ExtractDomainLabel-2 ~> .List => . ...</k>



rule <k> ExtractDomainLabel-read ~> T:Object
        ~> ( ListItem(D:TypeId) Ds:List => Ds )
        ...
    </k>
    <O-read>... T |-> Rt:Set ...</O-read>
    <D-read>... D |-> ( S:Set => intersectSet(S, Rt) ) ...</D-read>

rule <k> ExtractDomainLabel-read ~> _T:Object ~> .List => . ...</k>

rule <k> ExtractDomainLabel-write ~> T:Object
        ~> ( ListItem(D:TypeId) Ds:List => Ds )
```

```
            ...
        </k>
        <O-write>... T |-> Wt:Set ...</O-write>
        <D-write>... D |-> ( S:Set => intersectSet(S, Wt) ) ...</D-write>

    rule <k> ExtractDomainLabel-write ~> _:Object ~> .List => . ...</k>

endmodule


module CHECK-1
    imports LABEL-BUILD
    imports LIST
    imports COLLECTIONS
    imports K-EQUAL
    imports CONFIGURATION

    syntax Controller ::= "check1"
                        | "check1-again"
                        | "AccessRuleChecks"
                        | "AccessRuleChecks-1"
                        | "AccessRuleChecks-2"
                        | "AccessRuleChecks-2-read"
                        | "AccessRuleChecks-2-write"
                        | "AccessRuleChecks-3"
                        | "AccessRuleChecks-3-read"
                        | "AccessRuleChecks-3-write"
                        | "ConsistencyCheck"
                        | "find-wrong-allow"
                        | "find-wrong-allow-1"

    rule <k> check1
            => build-labels
            ~> AccessRuleChecks
            ~> ConsistencyCheck
            ~> find-wrong-allow
            ...
        </k>

    rule <k> check1-again
            => build-labels-again
            ~> AccessRuleChecks
            ~> ConsistencyCheck
            ~> find-wrong-allow
            ...
        </k>


    // access rule checks
    rule <k> AccessRuleChecks => AccessRuleChecks-1 ~> L ...</k>
        <rules> L:List </rules>
        <indirect-rules> _:Set => .Set </indirect-rules>

    // rule <k> AccessRuleChecks-1
    //         ~> (ListItem(neverallow _:TypeName _:TypeName : _:ClassName _:PermName) Rs:List
     => Rs)
    //     ...</k>

    rule <k> AccessRuleChecks-1
            ~> ListItem(allow T1:TypeId T2:TypeId : C:ClassId #token("read","Id"))
                Rs:List
            => #if Wt <=Set Wd
                #then .K
                #else AccessRuleChecks-2-read ~> T1 ~> Wt -Set Wd
                #fi
            ~> AccessRuleChecks-1 ~> Rs
            ...
        </k>
        <D-write>... T1 |-> Wd:Set ...</D-write>
        <O-write>... T2 : C |-> Wt:Set ...</O-write>
        // requires notBool Wt <=Set Wd


    rule <k> AccessRuleChecks-1
            ~> ListItem(allow T1:TypeId T2:TypeId : C:ClassId #token("write","Id"))
                Rs:List
            => #if Rt <=Set Rd
                #then .K
                #else AccessRuleChecks-2-write ~> T1 ~> Wt -Set Wd
                #fi
            ~> AccessRuleChecks-1 ~> Rs
            ...
        </k>
        <D-write>... T1 |-> Wd:Set ...</D-write>
```

```
    <D-read> ... T1 |-> Rd:Set ...</D-read>
    <O-write>... T2 : C |-> Wt:Set ...</O-write>
    <O-read> ... T2 : C |-> Rt:Set ...</O-read>

rule <k> AccessRuleChecks-1 ~> .List => . ...</k>


rule <k> AccessRuleChecks-2-read
        ~> D:TypeId ~> SetItem(D1:TypeId) Ds:Set
        => AccessRuleChecks-3-read ~> D ~> D1 ~> keys_list(W)
        ~> AccessRuleChecks-2-read ~> D ~> Ds:Set
        ...
    </k>
    <O-write> W:Map </O-write>

rule <k> AccessRuleChecks-2-write
        ~> D:TypeId ~> SetItem(D1:TypeId) Ds:Set
        => AccessRuleChecks-3-write ~> D ~> D1 ~> keys_list(W)
        ~> AccessRuleChecks-2-write ~> D ~> Ds:Set
        ...
    </k>
    <O-write> W:Map </O-write>

rule <k> AccessRuleChecks-2-read  ~> _D:TypeId ~> .Set => . ...</k>
rule <k> AccessRuleChecks-2-write ~> _D:TypeId ~> .Set => . ...</k>


rule <k> AccessRuleChecks-3-read ~> _:TypeId ~> _:TypeId
        ~> .List => . ...
    </k>
rule <k> AccessRuleChecks-3-read ~> D:TypeId ~> D1:TypeId
        ~> ( ListItem(T:TypeId : C:ClassId) Ts:List => Ts )
        ...
    </k>
    <indirect-rules>
        S:Set => (
            #if D in {W[T : C]}:>Set
            #then SetItem(allow D1 T : C #token("write", "Id"))
            #else .Set #fi
        ) |Set S
    </indirect-rules>
    <O-write> W:Map </O-write>

rule <k> AccessRuleChecks-3-write ~> _:TypeId ~> _:TypeId
        ~> .List => . ...
    </k>
rule <k> AccessRuleChecks-3-write ~> D:TypeId ~> D1:TypeId
        ~> ( ListItem(T:TypeId : C:ClassId) Ts:List => Ts )
        ...
    </k>
    <indirect-rules>
        S:Set => (
            #if D in {W[T : C]}:>Set
            #then SetItem(allow D1 T : C #token("read", "Id"))
            #else .Set #fi
        ) |Set S
    </indirect-rules>
    <O-write> W:Map </O-write>


// again if there are indirect rules
rule <k> ConsistencyCheck
        => #if size(S1) ==Int 0 #then . #else check1-again #fi
        ...
    </k>
    <indirect-rules> S1:Set => .Set </indirect-rules>
    <rules> L:List => Set2List(List2Set(L) |Set S1) </rules>
    <indirect-rules-result>
        S2:Set => S2 |Set S1
    </indirect-rules-result>

rule <k> find-wrong-allow => find-wrong-allow-1 ~> S ...</k>
    <indirect-rules-result> S:Set </indirect-rules-result>


rule <k> find-wrong-allow-1 ~> .Set => . ...</k>

rule <k> find-wrong-allow-1
        ~> ( SetItem(allow T1:TypeId T2:TypeId : C:ClassId P:PermId)
            S:Set => S )
        ...
    </k>
    <wrongallow>
        S1:Set => {
```

```
                #if P in {M[(T1 T2 C)] orDefault .Set}:>Set
                    |Set {M[(T1 *  C)] orDefault .Set}:>Set
                    |Set {M[(*  T2 C)] orDefault .Set}:>Set
                #then SetItem(allow T1 T2 : C P)
                #else .Set #fi
            }:>Set |Set S1
        </wrongallow>
        <neverallow-map> M:Map </neverallow-map>
endmodule


module CHECK-2
    imports CONFIGURATION
    imports COLLECTIONS

    syntax Controller ::= "check2"
                        | "coverage-1"

    rule <k> check2 => coverage-1 ~> Set2List(S) ... </k>
        <rules> _ => Set2List(S) </rules>
        <allow> S:Set </allow>
        <types> Ts:Set </types>
        <domain-labels>
            <D-read>  _ => .Map </D-read>
            <D-write> _ => .Map </D-write>
        </domain-labels>
        <object-type-labels>
            <O-read>  _ => .Map </O-read>
            <O-write> _ => .Map </O-write>
        </object-type-labels>
        <t1-unused> _ => Ts </t1-unused>
        <t2-unused> _ => Ts </t2-unused>
        // <object-noread>  _ => Ts </object-noread>
        // <object-nowrite> _ => Ts </object-nowrite>
        // <domain-noread>  _ => Ts </domain-noread>
        // <domain-nowrite> _ => Ts </domain-nowrite>


    // rule <k> coverage-1
    //         ~> ( ListItem(allow T1:TypeName T2:TypeName : _C:ClassName #token("read","Id"))
    //             Rs:List => Rs )
    //         ...
    //     </k>
    //     <object-noread> S1:Set => S1 -Set SetItem(T2) </object-noread>
    //     <domain-noread> S2:Set => S2 -Set SetItem(T1) </domain-noread>

    // rule <k> coverage-1
    //         ~> ( ListItem(allow T1:TypeName T2:TypeName : _C:ClassName #token("write","Id"))
    //             Rs:List => Rs )
    //         ...
    //     </k>
    //     <object-nowrite> S1:Set => S1 -Set SetItem(T2) </object-nowrite>
    //     <domain-nowrite> S2:Set => S2 -Set SetItem(T1) </domain-nowrite>

    rule <k> coverage-1
            ~> ( ListItem(allow T1:TypeId T2:TypeId : _:ClassId _:PermId )
                Rs:List => Rs )
            ...
        </k>
        <t1-unused> S1:Set => S1 -Set SetItem(T1) </t1-unused>
        <t2-unused> S2:Set => S2 -Set SetItem(T2) </t2-unused>

    // rule <k> coverage-1 ~> ListItem(neverallow _:TypeName _:TypeName : _:ClassName _:PermName
        ) Rs:List
    //         => coverage-1 ~> Rs
    //     ...</k>

    rule <k> coverage-1 ~> .List => . ... </k>
        <rules> _ => .List </rules>
endmodule
```