User Guide
Interactive Theorem Proving Framework V1.0
for Declarative Cloud Orchestration

Hiroyuki Yoshida

May 2017

1 Introduction

This interactive theorem proving framework aims to facilitate interactive proof development by allowing proof developers to devote their time to thinking through meaning of their own specifications and specific reasons why the specifications have some desired properties rather than reinventing how to construct proof scores or proving many specific versions of some common lemmas, thereby reducing overall development time.

The framework supports developers using mainly two means to reuse proof developments as follows:

1. Procedure how to develop proof scores to verify invariant and leads-to properties of cloud orchestration specifications.
   It prescribes what kind of set of proof goals should be sufficient to verify invariant and leads-to properties of specifications and guides how to split cases to develop proof case trees. The procedure supports proof developers to make the specific proof goals for their own specifications and to proceed case splitting until all the goals reduce to true.

2. Reusable lemmas that are commonly used in the course of verification for invariant and leads-to properties of cloud orchestration specifications.
   The lemmas are already proved in a general level of abstraction and can be reused simply by renaming included general terms to some specific ones without reproving.

The reason why the framework can provide such a development procedure and reusable lemmas is because it makes specific specifications adopt a general structure and behavior model for the domain and its representation. Proof developers can describe such specifications only by instantiating logical templates provided by the framework and adding small specific part of codes.

The reusable entities provided by the framework are four kinds of logic templates, a lot of general predicates/operators included in the templates, and a lot of lemmas about such predicates/operators which are proved in the general level of abstraction. The framework also provides general sufficient conditions to verify invariant and leads-to properties of cloud orchestration specifications and provides a procedure to develop proof trees whose goals are the sufficient conditions.

This user guide describes the general structure and behavior model, the reusable entities, and the proof development procedure.

All of the CafeOBJ codes of the framework and example proof scores can be downloaded at `https://github.com/yuki-yoshida/JAIST`.

2 Structure and Behavior Model

Cloud Orchestration is automation of operations such as set-up, scale-out, scale-in, or shutdown of cloud systems. In order to verify correctness of an automated operation of a cloud system, we need to model the structure of the target cloud system and the behavior of the operation.

Definition [structure model]: A *structure model* of a cloud system consists of several *classes* of *objects*.

Definition [object]: An *object* has a *type*, an *identifier*(ID), a *local state*, and possibly *links* to other objects.

Definition [class]: An object belongs to a *class* and thus a class is a set of objects. We assume this set consists of countably infinite objects each of which has its fixed ID and type. Local states or links of objects may be dynamically changed. A class specifies the set of possible types, the set of possible local states of its objects. A class also specifies how its objects link to other objects.

Definition [behavior model of an automated operation]: The *behavior model* of an automated operation of a cloud system is a *state transition system* in which a set of *transition rules* of states specifies the behavior. We say a *global state* as a state of the state machine in order to avoid the confusion with local states of objects. A global state is a finite set of objects each of which is included in some class. A transition rule makes a global state transit to another global state where local states or links of some objects are changed.

3 Representation of Structure Models

Users of the framework should design representations of their specific models in CafeOBJ language. A class should be represented as a CafeOBJ module that defines a sort of its objects, a

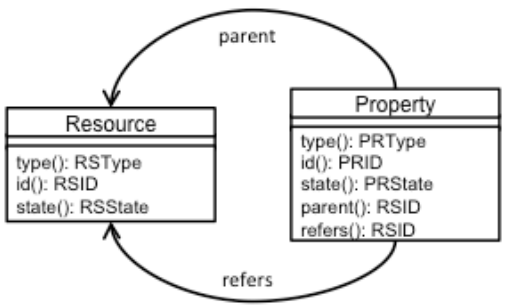

Figure 1: Example Classes

constructor of the sort, a set of literals of types, and a set of literals of local states. An object is represented as a ground constructor term of the sort.

Fig. 1 shows example classes, `Resource` and `Property` of a structure model. An object of each class has `type`, `id`, and `state` selectors. A `Property` object has a link to its parent `Resource` objects and one more link to its referring `Resource`, each of which corresponds additional selector `parent` and `refers` respectively.

The user can use a template module, `OBJECTBASE`, to easily define a module of a class only by adding small specific part of codes. A CafeOBJ module representing class `Resource` is typically named as `RESOURCE` and can be defined as follows:

```
module! RESOURCE {
  -- Instantiation of Template
  extending(OBJECTBASE
    * {sort Object -> Resource,
       sort ObjIDLt -> RSIDLt,
       sort ObjID -> RSID,
       sort ObjTypeLt -> RSTypeLt,
       sort ObjType -> RSType,
       sort ObjStateLt -> RSStateLt,
       sort ObjState -> RSState,
       sort SetOfObject -> SetOfResource,
       sort SetOfObjState -> SetOfRSState,
       op empObj -> empRS,
       op empState -> empSRS,
       op existObj -> existRS,
       op existObjInStates -> existRSInStates,
       op uniqObj -> uniqRS,
       op #ObjInStates -> #ResourceInStates,
       op getObject -> getResource,
       op allObjInStates -> allRSInStates,
       op allObjNotInStates -> allRSNotInStates,
       op someObjInStates -> someRSInStates}
  )

  -- Constructor
  -- res(RSType, RSID, RSState) is a Resource.
  op res : RSType RSID RSState -> Resource {constr}

  -- Variables
  var TRS : RSType
  var IDRS : RSID
  var SRS : RSState

  -- Selectors
  eq type(res(TRS,IDRS,SRS)) = TRS .
  eq id(res(TRS,IDRS,SRS)) = IDRS .
  eq state(res(TRS,IDRS,SRS)) = SRS .

  -- Local States
  ops initial started : -> RSStateLt {constr}
```

```
    eq (initial > started) = true .
  }
```

The following is a list of nine sorts predefined by template module `OBJECTBASE`:

- `Object` (renamed as `Resource` in this case)
  Sort for objects themselves.

- `ObjIDLt` (as `RSIDLt`)
  Subsort of `ObjID` for identifier literals. A literal is a constant for which `OBJECTBASE` predefines a special equality predicate such that _ = _ is exactly the same as _ == _ .

- `ObjID` (as `RSID`)
  Sort for identifiers of objects.

- `ObjTypeLt` (as `RSTypeLt`)
  Subsort of `ObjType` for type literals.

- `ObjType` (as `RSType`)
  Sort for types of objects.

- `ObjStateLt` (as `RSStateLt`)
  Subsort of `ObjState` for local state literals.

- `ObjState` (as `RSState`)
  Sort for local states of objects.

- `SetOfObject` (as `SetOfResource`)
  Soft for sets of objects.

- `SetOfObjState` (as `SetOfRSState`)
  Sort for sets of local states of objects.

The following is a list of part of operators predefined by template module `OBJECTBASE` whereas argument *obj* is an object, *id* is an identifier of an object, *seto* is a set of objects, and *setls* is a set of local states of objects:

- `empObj` (renamed as `empRS` in this case)
  Constant representing an empty set of objects.

- `empState` (as `empSRS`)
  Constant representing an empty set of local states of objects.

- `existObj` (as `existRS`)
  Predicate used as `existObj(`*seto*`,`*id*`)` which holds iff some object with identifier *id* is included in *seto*;
  $\exists o \in seto : \texttt{id}(o) = id$.

- `existObjInStates` (as `existRSInStates`)
  Predicate used as `existObjInStates(`*seto*`,`*id*`,`*setls*`)` which holds iff some object with identifier *id* is included in *seto* and its local state is included in *setls*;
  $\exists o \in seto : (\texttt{id}(o) = id \wedge \texttt{state}(o) \in setls)$.

- `uniqObj` (as `uniqRS`)
  Predicate used as `uniqObj(`*seto*`)` which holds iff the identifier of each object is unique in *seto*;
  $\forall o, o' \in seto : (o \neq o' \rightarrow \mathtt{id}(o) \neq \mathtt{id}(o'))$.

- `#ObjInStates` (as `#ResourceInStates`)
  Operator used as `#ObjInStates(`*setls*`,`*seto*`)` which returns the number of objects in *seto* whose local states are included in *setls*.

- `getObject` (as `getResource`)
  Operator used as `getObject(`*seto*`,`*id*`)` which returns an object in *seto* whose identifier is *id*.

- `allObjInStates` (as `allRSInStates`)
  Predicate used as `allObjInStates(`*seto*`,`*setls*`)` which holds iff the local states of all the objects in *seto* are included in *setls*;
  $\forall o \in seto : \mathtt{state}(o) \in setls$.

- `allObjNotInStates` (as `allRSNotInStates`)
  Predicate used as `allObjNotInStates(`*seto*`,`*setls*`)` which holds iff the local states of all the objects in *seto* are not included in *setls*;
  $\forall o \in seto : \mathtt{state}(o) \notin setls$.

- `someObjInStates` (as `someRSInStates`)
  Predicate used as `someObjInStates(`*seto*`,`*setls*`)` which holds iff there exists an objects in *seto* whose local state is included in *setls*;
  $\exists o \in seto : \mathtt{state}(o) \in setls$.

The module importing the instantiated template can extend it to freely define a constructor of objects and local state literals. In this case, module `RESOURCE` defines a constructor (`res`) of sort `Resource` whose arguments are a type, an identifier, and a local state of the resource.

It also defines two local state literals, `initial` and `started`, of a resource. Note that several proved lemmas provided by the framework ensure that some predefined predicate keeps to hold when a local state of an object changes by transitions. Such lemmas use the order $(\_ > \_)$ of local state literals in their conditions to decide the direction of transitions. Thus the user should define the order of literals when using such proved lemmas.

In addition, the module should implement three selector operators, `type`, `id`, and `state`, each of which takes a resource as an argument and returns the type, the identifier, and the local state of the resource respectively and `OBJECTBASE` uses them to implement the predefined general operators. Note that `OBJECTBASE` declares and uses these operators and so `RESOURCE` only should define them by equations.

Similarly, following module `PROPERTY` specifies class `Property`.

```
module! PROPERTY {
  protecting(RESOURCE)

  -- Instantiation of Template
  extending(OBJECTBASE
    * {sort Object -> Property,
       sort ObjIDLt -> PRIDLt,
```

```
        sort ObjID -> PRID,
        sort ObjTypeLt -> PRTypeLt,
        sort ObjType -> PRType,
        sort ObjStateLt -> PRStateLt,
        sort ObjState -> PRState,
        sort SetOfObject -> SetOfProperty,
        sort SetOfObjState -> SetOfPRState,
        op empObj -> empPR,
        op empState -> empSPR,
        op existObj -> existPR,
        op existObjInStates -> existPRInStates,
        op uniqObj -> uniqPR,
        op #ObjInStates -> #PropertyInStates,
        op getObject -> getProperty,
        op allObjInStates -> allPRInStates,
        op allObjNotInStates -> allPRNotInStates,
        op someObjInStates -> somePRInStates}
    )

    -- Constructor
    -- prop(PRType, PRID, PRState, RSID, RSID) is a Property.
    op prop : PRType PRID PRState RSID RSID -> Property {constr}

    -- Variables
    var TPR : PRType
    var IDPR : PRID
    var SPR : PRState
    vars IDRS1 IDRS2 : RSID

    -- Selectors
    op parent : Property -> RSID
    op refers : Property -> RSID
    eq type(prop(TPR,IDPR,SPR,IDRS1,IDRS2)) = TPR .
    eq id(prop(TPR,IDPR,SPR,IDRS1,IDRS2)) = IDPR .
    eq state(prop(TPR,IDPR,SPR,IDRS1,IDRS2)) = SPR .
    eq parent(prop(TPR,IDPR,SPR,IDRS1,IDRS2)) = IDRS1 .
    eq refers(prop(TPR,IDPR,SPR,IDRS1,IDRS2)) = IDRS2 .

    -- Local States
    ops notready ready : -> PRStateLt {constr}
    eq (notready > ready) = true .
  }
```

Module PROPERTY imports module RESOURCE using `protecting` because a property object links to its parent resource and also links to its referring resource.

Module PROPERTY defines a constructor (`prop`) of sort `Property` whose arguments are a type, an identifier, a local state, and links of the property. As noted before, a link is represented by an identifier of the linked object.

It also defines two local state literals, `notready` and `ready`, of a property and defines the order of them.

In addition to the mandatory selectors (`type`, `id`, and `state`), module `PROPERTY` declares and defines two more selectors, `parent` and `refers`, each of which returns a parent resource and a referring resource of the property respectively.

4 Instantiating Predefined Operators for Links

In addition to the operators provided by template module `OBJECTBASE`, two template modules `OBJLINKMANY2ONE` and `OBJLINKONE2ONE` provide many predefined operators/predicates for links between objects.

A template module `OBJLINKMANY2ONE` takes one parameter module of a class whose object links to another object. In order to provide predefined operators for links, the template module assumes that the parameter module defines eleven specific sorts and five specific operators. When the actual parameter module defines those sorts and operators with the different names from ones assumed, CafeOBJ allows to specify correspondence of the names. In the case of the example classes shown in Fig. 1, the sort for linking objects is `Property`, the sort for linked objects is `Resource`, and the selectors are `parent` and `refers` defined by module `PROPERTY`. The following module `LINKS` imports `OBJLINKMANY2ONE` twice for both kinds of links specifying the correspondence of the names:

```
module! LINKS {
  -- A Property links to its parent Resource
  extending(OBJLINKMANY2ONE(
    PROPERTY {sort Object -> Property,
              sort ObjID -> PRID,
              sort ObjType -> PRType,
              sort ObjState -> PRState,
              sort SetOfObject -> SetOfProperty,
              sort SetOfObjState -> SetOfPRState,
              sort LObject -> Resource,
              sort LObjID -> RSID,
              sort LObjState -> RSState,
              sort SetOfLObject -> SetOfResource,
              sort SetOfLObjState -> SetOfRSState,
              op link -> parent,
              op empLObj -> empRS,
              op existLObj -> existRS,
              op existLObjInStates -> existRSInStates,
              op getLObject -> getResource}
  )
  * {op hasLObj -> hasParent,
     op getXOfZ -> getRSOfPR,
     op getZsOfX -> getPRsOfRS,
     op getZsOfXInStates -> getPRsOfRSInStates,
     op getXsOfZs -> getRSsOfPRs,
     op getXsOfZsInStates -> getRSsOfPRsInStates,
     op getZsOfXs -> getPRsOfRSs,
     op getZsOfXsInStates -> getPRsOfRSsInStates,
     op allZHaveX -> allPRHaveRS,
     op allZOfXInStates -> allPROfRSInStates,
```

```
      op ifOfXThenInStates -> ifOfRSThenInStates,
      op ifXInStatesThenZInStates -> ifRSInStatesThenPRInStates}
  )

  -- A Property links to its referring Resource
  extending(OBJLINKMANY2ONE(
    PROPERTY {sort Object -> Property,
              sort ObjID -> PRID,
              sort ObjType -> PRType,
              sort ObjState -> PRState,
              sort SetOfObject -> SetOfProperty,
              sort SetOfObjState -> SetOfPRState,
              sort LObject -> Resource,
              sort LObjID -> RSID,
              sort LObjState -> RSState,
              sort SetOfLObject -> SetOfResource,
              sort SetOfLObjState -> SetOfRSState,
              op link -> refers,
              op empLObj -> empRS,
              op existLObj -> existRS,
              op existLObjInStates -> existRSInStates,
              op getLObject -> getResource}
    )
    * {op hasLObj -> hasRefRS,
       op getXOfZ -> getRRSOfPR,
       op getZsOfX -> getPRsOfRRS,
       op getZsOfXInStates -> getPRsOfRRSInStates,
       op getXsOfZs -> getRRSsOfPRs,
       op getXsOfZsInStates -> getRRSsOfPRsInStates,
       op getZsOfXs -> getPRsOfRRSs,
       op getZsOfXsInStates -> getPRsOfRRSsInStates,
       op allZHaveX -> allPRHaveRRS,
       op allZOfXInStates -> allPROfRRSInStates,
       op ifOfXThenInStates -> ifOfRRSThenInStates,
       op ifXInStatesThenZInStates -> ifRRSInStatesThenPRInStates}
    )
}
```

The following is a list of eleven sorts assumed by module `OBJLINKMANY2ONE`:

- `Object` (actually named as `Property` in this case)
  Sort for linking objects.

- `ObjID` (as `PRID`)
  Sort for identifiers of linking objects.

- `ObjType` (as `PRType`)
  Sort for types of linking objects.

- `ObjState` (as `PRState`)
  Sort for local states of linking objects.

- `SetOfObject` (as `SetOfProperty`)
  Sort for sets of linking objects.

- `SetOfObjState` (as `SetOfPRState`)
  Sort for sets of local states of linking objects.

- `LObject` (as `Resource`)
  Sort for linked objects.

- `LObjID` (as `RSID`)
  Sort for identifiers of linked objects.

- `LObjState` (as `RSState`)
  Sort for local states of linked objects.

- `SetOfLObject` (as `SetOfResource`)
  Sort for sets of linked objects.

- `SetOfLObjState` (as `SetOfRSState`)
  Sort for sets of local states of linked objects.

The following is a list of five operators assumed by module `OBJLINKMANY2ONE` whereas argument *obj* is a linking object, *lid* is an identifier of a linked object, *setlo* is a set of linked objects, and *setlls* is a set of local states of linked objects:

- `link` (actually named as `parent` and `refers` in this case)
  Selector used as `link(`*obj*`)` which returns the identifier of the object linked by *obj*.

- `empLObj` (as `empRS`)
  Constant representing an empty set of linked objects.

- `existLObj` (as `existRS`)
  Predicate used as `existLObj(`*setlo*`,`*lid*`)` which holds iff a linked object with identifier *lid* is included in *setlo*;
  $\exists lo \in setlo : \texttt{id}(lo) = lid$.

- `existLObjInStates` (as `existRSInStates`)
  Predicate used as `existLObjInStates(`*setlo*`,`*lid*`,`*setlls*`)` which holds iff a linked object with identifier *lid* is included in *setlo* and its local state is included in *setlls*;
  $\exists lo \in setlo : (\texttt{id}(lo) = lid \wedge \texttt{state}(lo) \in setlls)$.

- `getLObject` (as `getResource`)
  Operator used as `getLObject(`*setlo*`,`*lid*`)` which returns an object in *setlo* whose identifier is *lid*.

Note that `LINKS` imports `OBJLINKMANY2ONE` twice but only selector `link` is specified differently, `parent` and `refers`, and others are the same.

Many operators/predicates between linking (Z) and linked (X) objects are provided. In this case, each of them is twice renamed differently. The following is a list of part of operators predefined by template module `OBJLINKMANY2ONE` whereas argument *obj* is a linking object, *seto* is a set of linking objects, *setls* is a set of local states of linking objects, *lobj* is a linked object, *lid* is an identifier of a linked object, *setlo* is a set of linked objects, and *setlls* is a set of local states of linked objects:

- `hasLObj` (renamed as `hasParent` and `hasRefRS` in this case)
  Predicate used as `hasLObj(`*obj*`,`*setlo*`)` which holds iff the object linked by *obj* is included in *setlo*;
  $\exists lo \in setlo : \texttt{id}(lo) = \texttt{link}(obj)$.

- `getXOfZ` (as `getRSOfPR` and `getRRSOfPR`)
  Operator used as `getXOfZ(`*setlo*`,`*obj*`)` which returns an object linked by *obj* and included in *setlo*. When there is no such object in *setlo*, what it returns is undefined.

- `getZsOfX` (as `getPRsOfRS` and `getPRsOfRRS`)
  Operator used as `getZsOfX(`*seto*`,`*lobj*`)` which returns a subset *seto* each of whose element object links to *lobj*.

- `getZsOfXInStates` (as `getPRsOfRSInStates` and `getPRsOfRRSInStates`)
  Operator used as `getZsOfXInStates(`*seto*`,`*lobj*`,`*setls*`)` which returns a subset of *seto* each of whose element object links to *lobj* and is in one of local states of *setls*.

- `getXsOfZs` (as `getRSsOfPRs` and `getRRSsOfPRs`)
  Operator used as `getXsOfZs(`*setlo*`,`*seto*`)` which returns a subset of *setlo* each of whose element object is linked by some object included in *seto*.

- `getXsOfZsInStates` (as `getRSsOfPRsInStates` and `getRRSsOfPRsInStates`)
  Operator used as `getXsOfZsInStates(`*setlo*`,`*seto*`,`*setlls*`)` which returns a subset of *setlo* each of whose element object is linked by some object included in *seto* and is in one of local states of *setlls*.

- `getZsOfXs` (as `getPRsOfRSs` and `getPRsOfRRSs`)
  Operator used as `getZsOfXs(`*seto*`,`*setlo*`)` which returns a subset of *seto* each of whose element object links to some object included in *setlo*.

- `getZsOfXsInStates` (as `getPRsOfRSsInStates` and `getPRsOfRRSsInStates`)
  Operator used as `getZsOfXsInStates(`*seto*`,`*setlo*`,`*setls*`)` which returns a subset of *seto* each of whose element object links to some object included in *setlo* and is in one of local states of *setls*.

- `allZHaveX` (as `allPRHaveRS` and `allPRHaveRRS`)
  Predicate used as `allZHaveX(`*seto*`,`*setlo*`)` which holds iff every object included in *seto* has objects linked by it which are included in *setlo*;
  $\forall o \in seto, \exists lo \in setlo : \texttt{id}(lo) = \texttt{link}(o)$.

- `allZOfXInStates` (as `allPROfRSInStates` and `allPROfRRSInStates`)
  Predicate used as `allZOfXInStates(`*seto*`,`*lid*`,`*setls*`)` which holds iff every object included in *seto* whose link is *lid* is in one of locals state in *setls*;
  $\forall o \in seto : (\texttt{link}(o) = lid \rightarrow \texttt{state}(o) \in setls)$.

- `ifOfXThenInStates` (as `ifOfRSThenInStates` and `ifOfRRSThenInStates`)
  Predicate used as `ifOfXThenInStates(`*obj*`,`*lid*`,`*setls*`)` which holds iff the link of *obj* is not *lid* or the local state of *obj* is included in *setls*;
  $\texttt{link}(obj) = lid \rightarrow \texttt{state}(obj) \in setls$.

- `ifXInStatesThenZInStates`
  (as `ifRSInStatesThenPRInStates` and `ifRRSInStatesThenPRInStates`)
  Predicate used as `ifXInStatesThenZInStates`(*setlo*, *setlls*, *seto*, *setls*) which holds iff every object included in *setlo* whose local sate is included in *setlls* is linked by objects included in *seto* each of which is in one of local states in *setls*;

$$\forall lo \in setlo : (\texttt{state}(lo) \in setlls \rightarrow \forall o \in seto : (\texttt{link}(o) = \texttt{id}(lo) \rightarrow \texttt{state}(o) \in setls)).$$

Similarly template module `OBJLINKONE2ONE` provides predicates for one to one relationships between objects. The following is a list of operators predefined by `OBJLINKONE2ONE` whereas argument *obj* is a linking object, *seto* is a set of linking objects, *setls* is a set of local states of linking objects, *lobj* is a linked object, *lid* is an identifier of a linked object, *setlo* is a set of linked objects, and *setlls* is a set of local states of linked objects:

- `existX` (renamed as `existCP` and `existRQ`)
  Predicate used as `existX`(*seto*,*lid*) which holds iff some object whose link is *lid* is included in *seto*;
  $\exists o \in seto : \texttt{link}(o) = lid$.

- `getXOfY` (as `getCPOfRL` and `getRQOfRL`)
  Operator used as `getXOfY`(*setlo*,*obj*) which returns an object linked by *obj* and included in *setlo*.

- `getXsOfYs` (as `getCPsOfRLs` and `getRQsOfRLs`)
  Operator used as `getXsOfYs`(*setlo*,*seto*) which returns a subset of *setlo* each of whose element object is linked by some object included in *seto*.

- `getXsOfYsInStates` (as `getCPsOfRLsInStates` and `getRQsOfRLsInStates`)
  Operator used as `getXsOfYsInStates`(*setlo*,*seto*,*setlls*) which returns a subset of *setlo* each of whose element object is linked by some object included in *seto* and is in one of local states of *setlls*.

- `getYOfX` (as `getRLOfCP` and `getRLOfRQ`)
  Operator used as `getYOfX`(*seto*,*lobj*) which returns an object which included in *seto* and whose link is *lobj*.

- `getYsOfXs` (as `getRLsOfCPs` and `getRLsOfRQs`)
  Operator used as `getYsOfXs`(*seto*,*setlo*) which returns a subset of *seto* each of whose element object links to some object included in *setlo*.

- `getYsOfXsInStates` (as `getRLsOfCPsInStates` and `getRLsOfRQsInStates`)
  Operator used as `getYsOfXsInStates`(*seto*,*setlo*,*setls*) which returns a subset of *seto* each of whose element object links to some object included in *setlo* and is in one of local states of *setls*.

- `uniqX` (as `uniqCP` and `uniqRQ`)
  Predicate used as `uniqX`(*seto*) which holds iff the link of each object is unique in *seto*;
  $\forall o, o' \in seto : (o \neq o' \rightarrow \texttt{link}(o) \neq \texttt{link}(o'))$.

- `YOfXInStates` (as `RLOfCPInStates` and `RLOfRQInStates`)
  Predicate used as `YOfXInStates(`*seto*`,`*lid*`,`*setls*`)` which holds iff an object included in *seto* whose link is *lid* is in one of locals state in *setls*;
  $\exists o \in seto : (\texttt{link}(o) = lid \wedge \texttt{state}(o) \in setls)$.

- `ifXInStatesThenYInStates`
  (as `ifCPInStatesThenRLInStates` and `ifRQInStatesThenRLInStates`)
  Predicate used as `ifXInStatesThenYInStates(`*setlo*`,`*setlls*`,`*seto*`,`*setls*`)` which holds iff every object included in *setlo* whose local sate is included in *setlls* is linked by an object included in *seto* which is in one of local states in *setls*;
  $\forall lo \in setlo : (\texttt{state}(lo) \in setlls \rightarrow \exists o \in seto : (\texttt{link}(o) = \texttt{id}(lo) \wedge \texttt{state}(o) \in setls))$.

- `ifYInStatesThenXInStates`
  (as `ifRLInStatesThenCPInStates` and `ifRLInStatesThenRQInStates`)
  Predicate used as `ifYInStatesThenXInStates(`*seto*`,`*setls*`,`*setlo*`,`*setlls*`)` which holds iff every object included in *seto* whose local sate is included in *setls* links to an object included in *setlo* which is in one of local states in *setlls*;
  $\forall o \in seto : (\texttt{state}(o) \in setls \rightarrow \exists lo \in setlo : (\texttt{link}(o) = \texttt{id}(lo) \wedge \texttt{state}(lo) \in setlls))$.

- `allYHaveX` (as `allRLHaveCP` and `allRLHaveRQ`)
  Predicate used as `allYHaveX(`*seto*`,`*setlo*`)` which holds iff every object included in *seto* has an object linked by it which is included in *setlo*;
  $\forall o \in seto, \exists lo \in setlo : \texttt{id}(lo) = \texttt{link}(o)$.

- `allXHaveY` (as `allCPHaveRL` and `allRQHaveRL`)
  Predicate used as `allXHaveY(`*setlo*`,`*seto*`)` which holds iff every object included in *setlo* has an object which links to it and is included in *seto*;
  $\forall lo \in setlo, \exists o \in seto : \texttt{id}(lo) = \texttt{link}(o)$.

- `OnlyOneYOfX` (renamed as `onlyOneRLOfCP` and `onlyOneRLOfRQ`)
  Predicate used as `OnlyOneYOfX(`*seto*`,`*lid*`)` which holds iff only one object whose link is *lid* is included in *seto*;
  $\exists o \in seto : \texttt{link}(o) = lid \wedge (\forall o' \in seto : o \neq o' \rightarrow \texttt{link}(o) \neq \texttt{link(o')})$.

5 Representation of Behavior Models

A global state is represented in CafeOBJ as a ground constructor term of sort `State`. The user should define the constructor, which is typically a tuple of sets of objects, each of the sets is a finite subset of a class.

In the case of the example classes shown in Fig. 1, sort `State` is typically defined as a pair of a set of resources and a set of properties as follows:

```
module! STATE {
  protecting(LINKS)
  [State]
  op <_,_> : SetOfResource SetOfProperty -> State {constr}
}
```

The behavior is modeled and represented by a set of two transition rules. The following is an example of such a transition rule:

```
module! STATERules {
 protecting(STATEfuns)

 -- Variables
 vars IDRS IDRRS : RSID
 var IDPR : PRID
 var TRS : RSType
 var TPR : PRType
 var SetRS : SetOfResource
 var SetPR : SetOfProperty

 -- Start an initial resource
 --  if all of its properties are ready.
 ctrans [R01]:
    < (res(TRS,IDRS,initial) SetRS), SetPR >
 => < (res(TRS,IDRS,started) SetRS), SetPR >
    if allPROfRSInStates(SetPR,IDRS,ready) .
}
```

As described above, predicate `allPROfRSInStates(SetPR,IDRS,ready)` checks a set of properties `SetPR` whether every property of resource `IDRS` is `ready`. Thus, rule `R01` means that an `initial` resource becomes `started` when all of its properties are `ready`.

6 Proved Lemmas for Predefined Predicates

In the course of verification, a lot of lemmas about predefined predicates are commonly required. The framework provides many typical lemmas which are already proved in a general level of abstraction and can be used for any instantiated predicates without individual reproving.

6.1 Basic Lemmas

Implication Lemma: Let `A` and `B` be Boolean terms in CafeOBJ , then "`A implies B`" is equivalent to "`A and B = A` ."

A lemma typically has a form $A \rightarrow B$. When using this to prove a *goal*, we may write a proof score in CafeOBJ as follows:

```
reduce (A implies B) implies goal .
```

This style is not only complicated but also very expensive to execute by CafeOBJ system. Using the Implication Lemma, we can define lemmas in an independent style from goals as follows:

```
eq (A1 and B1) = A1 .
eq (A2 and B2) = A2 .
...
:goal { eq goal = true . }
```

Set Predicate Lemma: Let `S` be a set of object, `P` a predicate of an object, `allObjP` a predicate of a set of objects where `allObjP(S)` holds iff `P(O)` holds for every object `O` in `S`. Then, if `allObjP(S)` does not hold, then there exists an object `O'` and a set `S'` of objects such that `S=(O'`

S') holds and P(O') does not hold.

Corollary: Let S be a set of object, P a predicate of an object, someObjP a predicate of a set of objects where someObjP(S) holds iff P(O) holds for some object O in S. Then, if someObjP(S) holds, then there exists an object O' and a set S' of objects such that S=(O' S') holds and P(O') holds.

Template module OBJECTBASE predefines a general predicate allObjP that uses an object predicate P and checks if P(O) holds for every object O in a given set of objects. Similarly it predefines a general predicate someObjP. Here, it is important to note that many predicates provided by the template modules are ones instantiated from allObjP or someObjP.

For example, allZOfXInStates is instantiated from allObjP where P(O) holds iff O is in one of given local states whenever it links to a given linked object. As explained above, allPROfRSInStates is renamed from allZOfXInStates and thus the Set Predicate Lemma can be used to split cases where the condition of rule R01 does or does not hold as follows:

```
:csp {
  eq allPROfRSInStates(setPR,idRS,ready) = true .
  eq setPR = (PR' setPR') .
}
```

Note that in this case, PR' should be a property whose parent is resource idRS but is not ready (i.e. is notready). Thus, PR' can be represented as prop(tpr,idPR,notready,idRS,idRRS) where tpr, idPR, and idRRS are arbitrary constants. Then, the following case splitting collectively covers all of the cases:

```
:csp {
  eq allPROfRSInState(setPR,idRS,ready) = true .
  eq setPR = (prop(tpr,idPR,notready,idRS,idRRS) setPR') .
}
```

For another example, since existRS is instantiated from someObjP, a typical case splitting code is as follows:

```
:csp {
  eq existRS(setRS,idRS) = false .
  eq setRS = (res(trs,idRS,srs) setRS') .
}
```

The following is a list of proved lemmas for predefined set predicates:

```
-- When O is included in S, there exist some S' such that S = (O S').
eq set-lemma00(S, O)
   = (O \in S) implies (S = (O s'(S))) .

-- When allObjP(S,A) does not hold, there exist some E and S'
-- such that S = (E S') and p(E,A) does not hold.
eq set-lemma01(S, A)
   = not allObjP(S,A) implies (S = (e(S) s'(S))) and not p(e(S),A) .

-- When onlyOneObjP(S,A) holds, there exist some E and S'
-- such that S = (E S') and p(E,A) holds.
```

```
eq set-lemma02(S, A)
   = onlyOneObjP(S,A) implies
     (S = (e(S) s'(S))) and p(e(S),A) and allObjNotP(s'(S),A).

-- someObjP(S,A) if and only if not allObjNotP(S,A).
eq set-lemma03(S, A)
   = someObjP(S,A) iff not allObjNotP(S,A) .

-- When p(O,A) holds and O is included in S, then someObjP(S,A) holds.
eq set-lemma04(S, O, A)
   = p(O,A) and (O \in S) implies someObjP(S,A) .

-- When onlyOneObjP(S,A) holds, someObjP(S,A) also holds.
eq set-lemma05(S, A)
   = onlyOneObjP(S,A) implies someObjP(S,A) .

-- When p(O,A) holds, allObjP((O S),A) is equivalent to allObjP(S,A).
eq set-lemma06(O, S, A)
   = allObjP((O S),A) implies allObjP(S,A) .

-- When O \in S and O1 \in O holds, O1 \in S also holds.
eq set-lemma07(O,O1,S)
   = O1 \in O implies O1 \in S
   when O \in S .

-- When O \in S holds, O \in (S S') also holds.
eq set-lemma08(O,S,S')
   = O \in S implies O \in (S S') .

-- When subset(S,S') holds, subset(S,(O S')) also holds.
eq set-lemma09(O,S,S')
   = subset(S,S') implies subset(S,(O S')) .

-- When subset(S1,S) and subset(S2,S) holds, subset((S1 S2),S) also holds.
eq set-lemma10(S,S1,S2)
   = subset(S2,S) implies subset((S1 S2),S)
   when subset(S1,S) .

-- When subset(S,S1) holds, subset(S,(S1 S2)) also holds.
eq set-lemma11(S,S1,S2)
   = subset(S,S1) implies subset(S,(S1 S2)) .

-- subset(S,S)) alway holds.
eq set-lemma12(S)
   = subset(S,S) .
```

6.2 Lemmas for Link Predicates

The framework provides many proved lemmas for predefined predicates provided by template OBJLINKMANY2ONE and OBJLINKONE2ONE. They can be used for any instantiated predicates

without individual reproving similarly as the basic lemmas. The following is a list of proved lemmas for predefined link predicates:

```
eq o2o-lemma01(IDX,S_Y,St_Y)
   = allObjInStates(S_Y,St_Y) implies YOfXInStates(S_Y,IDX,St_Y)
   when uniqX(S_Y) and onlyOneYOfX(S_Y,IDX) .

eq o2o-lemma02(Y,Y',S_X,St_X,S_Y,St_Y)
   = ifXInStatesThenYInStates(S_X,St_X,(Y S_Y),St_Y) implies
       ifXInStatesThenYInStates(S_X,St_X,(Y' S_Y),St_Y)
   when changeObjState(Y,Y') and (state(Y') \in St_Y) .

eq o2o-lemma03(S_X,St_X,S_Y,St_Y)
   = allObjInStates(S_Y,St_Y) implies
       ifXInStatesThenYInStates(S_X,St_X,S_Y,St_Y)
   when uniqX(S_Y) and allXHaveY(S_X,S_Y) .

eq o2o-lemma04(Y,Y',S_X,St_X,S_Y,St_Y)
   = ifXInStatesThenYInStates(S_X,St_X,(Y S_Y),St_Y) implies
       ifXInStatesThenYInStates(S_X,St_X,(Y' S_Y),St_Y)
   when not existLObj(S_X,link(Y)) and changeObjState(Y,Y') .

eq o2o-lemma05(Y,Y',S_X,St_X,S_Y,St_Y)
   = ifXInStatesThenYInStates(S_X,St_X,(Y S_Y),St_Y) implies
       ifXInStatesThenYInStates(S_X,St_X,(Y' S_Y),St_Y)
   when changeObjState(Y,Y') and not (state(Y) \in St_Y) .

eq m2o-lemma01(IDX,S_Z,SZ,St_Z)
   = allObjInStates(S_Z,SZ) implies allZOfXInStates(S_Z,IDX,St_Z)
   when (SZ \in St_Z) .

eq m2o-lemma02(IDX,S_Z,SZ,OT,St_Z)
   = allObjInStates(S_Z,SZ) implies
       allZOfTypeOfXInStates(S_Z,OT,IDX,St_Z)
   when (SZ \in St_Z) .

eq m2o-lemma03(S_Z,Z,St_Z)
   = allZOfXInStates(S_Z,link(Z),St_Z) implies (state(Z) \in St_Z)
   when (Z \in S_Z) .

eq m2o-lemma04(S_Z,IDX,SZ,St_Z)
   = (getZsOfXInStates(S_Z,IDX,SZ) = empObj) implies
       allZOfXInStates(S_Z,IDX,St_Z)
   when allZOfXInStates(S_Z,IDX,(SZ St_Z)) .

eq m2o-lemma05(X,X',S_Z,S_X)
   = allZHaveX(S_Z,(X S_X)) implies allZHaveX(S_Z,(X' S_X))
  when changeLObjState(X,X') .

eq m2o-lemma06(S_X,St_X,S_Z,SZ,St_Z)
   = allObjInStates(S_Z,SZ) implies
```

```
        ifXInStatesThenZInStates(S_X,St_X,S_Z,St_Z)
   when (SZ \in St_Z) .

eq m2o-lemma07(S_X,SX,St_X,S_Z,St_Z)
   = allObjInStates(S_X,SX) implies
        ifXInStatesThenZInStates(S_X,St_X,S_Z,St_Z)
   when not (SX \in St_X) .

eq m2o-lemma08(Z,Z',S_X,St_X,S_Z,St_Z)
   = ifXInStatesThenZInStates(S_X,St_X,(Z S_Z),St_Z) implies
        ifXInStatesThenZInStates(S_X,St_X,(Z' S_Z),St_Z)
   when not existLObj(S_X,link(Z)) and changeObjState(Z,Z') .

eq m2o-lemma09(S_X,St_X,Z,S_Z,St_Z)
   = ifXInStatesThenZInStates(S_X,St_X,(Z S_Z),St_Z) implies
        ifXInStatesThenZInStates(S_X,St_X,S_Z,St_Z) .

eq m2o-lemma10(X,X',S_X,St_X,S_Z,St_Z)
   = ifXInStatesThenZInStates((X S_X),St_X,S_Z,St_Z) implies
        ifXInStatesThenZInStates((X' S_X),St_X,S_Z,St_Z)
   when not (state(X') \in St_X) and changeLObjState(X,X') .

eq m2o-lemma11(Z,Z',S_X,St_X,S_Z,St_Z)
   = ifXInStatesThenZInStates(S_X,St_X,(Z S_Z),St_Z) implies
        ifXInStatesThenZInStates(S_X,St_X,(Z' S_Z),St_Z)
   when (state(Z') \in St_Z) and changeObjState(Z,Z') .

eq m2o-lemma12(X,X',S_X,St_X,S_Z,St_Z)
   = ifXInStatesThenZInStates((X S_X),St_X,S_Z,St_Z) implies
        ifXInStatesThenZInStates((X' S_X),St_X,S_Z,St_Z)
   when allZOfXInStates(S_Z,id(X'),St_Z) and changeLObjState(X,X') .

eq m2o-lemma13(Z,Z',S_X,St_X,S_Z,St_Z)
   = ifXInStatesThenZInStates(S_X,St_X,(Z S_Z),St_Z) implies
        ifXInStatesThenZInStates(S_X,St_X,(Z' S_Z),St_Z)
   when not (state(Z) \in St_Z) and changeObjState(Z,Z') .

eq m2o-lemma14(S_X,St_X,S_Z,SZ,OT,St_Z)
   = allObjInStates(S_Z,SZ) implies
        ifXInStatesThenZOfTypeInStates(S_X,St_X,S_Z,OT,St_Z)
   when (SZ \in St_Z) .

eq m2o-lemma15(S_X,SX,St_X,S_Z,OT,St_Z)
   = allObjInStates(S_X,SX) implies
        ifXInStatesThenZOfTypeInStates(S_X,St_X,S_Z,OT,St_Z)
   when not (SX \in St_X) .

eq m2o-lemma16(Z,Z',S_X,St_X,S_Z,OT,St_Z)
   = ifXInStatesThenZOfTypeInStates(S_X,St_X,(Z S_Z),OT,St_Z) implies
        ifXInStatesThenZOfTypeInStates(S_X,St_X,(Z' S_Z),OT,St_Z)
```

```
   when not existLObj(S_X,link(Z)) and changeObjState(Z,Z') .

eq m2o-lemma17(S_X,St_X,Z,S_Z,OT,St_Z)
   = ifXInStatesThenZOfTypeInStates(S_X,St_X,(Z S_Z),OT,St_Z) implies
       ifXInStatesThenZOfTypeInStates(S_X,St_X,S_Z,OT,St_Z) .

eq m2o-lemma18(X,X',S_X,St_X,S_Z,OT,St_Z)
   = ifXInStatesThenZOfTypeInStates((X S_X),St_X,S_Z,OT,St_Z) implies
       ifXInStatesThenZOfTypeInStates((X' S_X),St_X,S_Z,OT,St_Z)
   when not (state(X') \in St_X) and changeLObjState(X,X') .

eq m2o-lemma19(Z,Z',S_X,St_X,S_Z,OT,St_Z)
   = ifXInStatesThenZOfTypeInStates(S_X,St_X,(Z S_Z),OT,St_Z) implies
       ifXInStatesThenZOfTypeInStates(S_X,St_X,(Z' S_Z),OT,St_Z)
   when (state(Z') \in St_Z) and changeObjState(Z,Z') .

eq m2o-lemma20(X,X',S_X,St_X,S_Z,OT,St_Z)
   = ifXInStatesThenZOfTypeInStates((X S_X),St_X,S_Z,OT,St_Z) implies
       ifXInStatesThenZOfTypeInStates((X' S_X),St_X,S_Z,OT,St_Z)
   when allZOfTypeOfXInStates(S_Z,OT,id(X'),St_Z) and
        changeLObjState(X,X') .

eq m2o-lemma21(S_X,St_X,Z,Z',S_Z,OT,St_Z)
   = ifXInStatesThenZOfTypeInStates(S_X,St_X,(Z S_Z),OT,St_Z) implies
       ifXInStatesThenZOfTypeInStates(S_X,St_X,(Z' S_Z),OT,St_Z)
   when not (type(Z) = OT) and changeObjState(Z,Z') .

eq m2o-lemma22(Z,S_Z,S_X,St_X)
   = getLObject(S_X,link(Z)) \in getXsOfZsInStates(S_X,S_Z,St_X)
   when not S_X = empLObj and
        (state(getLObject(S_X,link(Z))) \in St_X) and (Z \in S_Z) .

eq m2o-lemma23(S_Z,S_Z',S_X,St_X)
   = subset(S_Z,S_Z') implies
       subset(getXsOfZsInStates(S_X,S_Z,St_X),
              getXsOfZsInStates(S_X,S_Z',St_X)) .

eq m2o-lemma24(X,X',S_X,S_Z,St_X)
   = subset(getXsOfZsInStates((X' S_X),S_Z,St_X),
            getXsOfZsInStates((X S_X),S_Z,St_X))
   when changeLObjState(X,X') and not state(X') \in St_X .

eq m2o-lemma25(X,S_X,S_Z,St_Z)
   = subset(getZsOfXInStates(S_Z,id(X),St_Z),
            getZsOfXsInStates(S_Z,S_X,St_Z))
   when X \in S_X .

eq m2o-lemma26(S_X,S_X',S_Z,St_Z)
   = subset(S_X,S_X') implies
       subset(getZsOfXsInStates(S_Z,S_X,St_Z),
```

```
              getZsOfXsInStates(S_Z,S_X',St_Z)) .

eq m2o-lemma27(Z,Z',S_Z,S_X,St_Z)
   = subset(getZsOfXsInStates((Z' S_Z),S_X,St_Z),
            getZsOfXsInStates((Z S_Z),S_X,St_Z))
   when changeObjState(Z,Z') and not state(Z') \in St_Z .

eq m2o-lemma28(Z,Z',S_Z,X,St_Z)
   = subset(getZsOfXInStates((Z' S_Z),X,St_Z),
            getZsOfXInStates((Z S_Z),X,St_Z))
   when changeObjState(Z,Z') and not state(Z') \in St_Z .
```

6.3 Cyclic Dependency Lemma

A rule typically produces dependency of objects. For example, rule `R01` above makes a resource transit from `initial` to `started` when all of its properties are `ready`, which means the resource depends on its properies. If such dependency is cyclic it should be troublesome because there may be a situation where each of objects in the cycle is waiting for its dependent object and no rule is applicable to any of them.

In order to start transitions and reach a desired final state, a cloud system should not include such cyclic dependency. Verification of the system requires (1) to formalize that the dependency is acyclic, (2) to prove the invariant property of the acyclicness, and (3) to prove that when acyclic there exists at least one applicable transition rule and the state machine continues to transit. The framework provides a template module to formalize acyclicness of dependency for (1) and a lemma that guarantees existence of applicable rules for (3). It also provides several common techniques and proved lemmas for (2).

6.3.1 Definitions of Dependency and Acyclicness

This section describes an intuitive definition of cyclic dependency.

Definition [pre-transit local state set]: The *pre-transit local state set* of transition rule R for class C, denoted $prels(R, C)$, is the set of local states of C where $s \in prels(R, C)$ iff there exists some local state s' of C such that R can make an object of C transit from s to s'.

For example, $prels($`R01`$,$ `Resource`$)$ is { `initial` } because `R01` can make a resource transit from `initial` to `started`.

Definition [depends on]: Suppose that global state S includes objects X and X' and transition rule R cannot make X transit in S. We say X *depends on* X' *in* S *w.r.t.* R, denoted $dep_R(X, X', S)$, iff R can make X transit in some S' where S' is a minimally different global state of S and the local state of X' in S is different from that in S'. Here we say "minimally different" to mean that there exists no global state S'' such that R can make X transit in S'' and difference between S and S'' is smaller than that between S and S'. We also say X *depends on* X' *in* S, denoted $dep(X, X', S)$, when there exists some transition rule R such that $dep_R(X, X', S)$.

For example, when some of properties of an initial resource are `notready` in global state S, the resource depends on the properties in S w.r.t. `R01`, because `R01` can make the resource transit

when all of its `notready` properties become `ready`.

Definition [depending set]: The *depending set* of object X in global state S, denoted $DS(X, S)$, is recursively defined as (1) if X depends on some other object X' in S then X' is included in $DS(X, S)$, and (2) if $X' \in DS(X, S)$ and X' depends on some other object X'' in S then X'' is included in $DS(X, S)$.

Definition [no cyclic dependency]: We say object X *is in no cyclic dependency in global state* S, denoted $noCycle(X, S)$, iff X itself is not included in $DS(X, S)$. We also say there is *no cyclic dependency of class* C *in* S, denoted $noCycle_C(S)$, iff all of the objects of C in S are in no cyclic dependency in S.

Definition [dependency chain]: Let $X_1, \ldots, X_n$ be objects, and S a global state, then the *dependency chain* in S, denoted $dc([X_1, \ldots, X_n], S)$, is defined as $\forall i \in [1, n-1] : dep(X_i, X_{i+1}, S)$.

Definition [directly depending set]: Let C be a class of object X. The *directly depending set of the same class as* X *in global state* S, denoted $DDS_C(X, S)$, is defined as
$\{ X' \mid \exists dc([X, X_1, \ldots, X_n, X'], S) \wedge X' \in C \wedge \forall i \in [1, n] : X_i \notin C \}$. We also say X *directly depends on* X' *in* S when $X' \in DDS_c(X, S)$.

When X and X' are objects of calss C, $X' \in DDS_C(X)$ means that there exists a dependency chain in which the first object is X, the last object is X', and every object between X and X' is not of C.

6.3.2 Formalization of Acyclicness

Using the formalization of cyclic dependency explained above, the framework provides a predicate, `noCycleC(S)`, which checks there is no cyclic dependency in the given global state S. Predicate `noCycleC` is defined by a template module, `CYCLEPRED` and a parameter module, `PRMCYCLE`:

```
module* PRMCYCLE {
  [Object < SetOfObject]
  op empObj : -> SetOfObject
  op _ _ : SetOfObject SetOfObject -> SetOfObject
  op _\in_ : Object SetOfObject -> Bool

  [State]
  op getAllObjInState : State -> SetOfObject
  op DDSC : Object State -> SetOfObject
}

module! CYCLEPRED(P :: PRMCYCLE) {
  var O  : Object
  vars V OS : SetOfObject
  var S : State

  pred noCycleC : State
  pred noCycleC : Object State
```

```
  pred noCycleC : SetOfObject SetOfObject State

  eq noCycleC(S)
     = noCycleC(getAllObjInState(S),empObj,S) .

  eq noCycleC(O,S)
     = noCycleC(O,empObj,S) .

  eq noCycleC(empObj,V,S)
     = true .
  eq noCycleC((O OS),V,S)
     = if O \in V then false else noCycleC(DDSC(O,S),(O V),S) fi
       and noCycleC(OS,V,S) .
}
```

Parameter module PRMCYCLE requires five operator parameters three of which can be defined just by using template module OBJECTBASE. The user of the framework should appropriately define getAllObjInState and DDSC because they are specific to each problem. Given a global state S, operator getAllObjInState(S) should return the set of all the objects of the specific class we focus. Operator DDSC(O,S) should return the directly depending set of the same class as the given object O in the given global state S.

Using these operators, template module CYCLEPRED defines predicate noCycleC. Given a global state S, predicate noCycleC(S) transitively visits objects in directly depending sets DDSC(O,S) and checks not to find any objects already visited. Template module CYCLEPRED can be instantiated as follows:

```
extending(CYCLEPRED(
      STATECyclefuns {sort Object -> Resource,
                      sort SetOfObject -> SetOfResource,
                      op empObj -> empRS,
                      op getAllObjInState -> getAllRSInState,
                      op DDSC -> DDSRS})
      * {op noCycleC -> noRSCycle}
      )
```

Resource, SetOfResource, empRS, getAllRSInState, and DDSRS are specified as actual parameters. noCycleC is renamed as noRSCycle.

6.3.3 Proof of Invariant Property of Acyclicness

In order to verify an automated cloud operation, the user of the framework should prove the invariant property of $noCycle_C$, especially should prove that $noCycle_c(S) \rightarrow noCycle_C(S')$ for any global state S and any possible next state S' of S. Although such proof is specific to each problem, there are several common techniques and the framework provides several proved lemmas for them.

It is often the case where a transition rule decreases dependencies between objects when it is applied. If the depending set become smaller than in the previous global states, $noCycle_c(S)$ keeps to hold.

Depending Subset Lemma: Let S and S' be global states and X an object in both of them. If $DS(X, S') \subseteq DS(X, S)$, then $noCycle(X, S) \rightarrow noCycle(X, S')$.

Corollary: Let C be a class of objects and let S and S' be global states. If $DS_C(X, S') \subseteq DS_C(X, S)$ for all objects X of C in S, then $noCycle_C(S) \rightarrow noCycle_C(S')$.

Corollary: Let C be a class of objects and let S and S' be global states. If $DDS_C(X, S') \subseteq DDS_C(X, S)$ for all objects X of C in S, then $noCycle_C(S) \rightarrow noCycle_C(S')$.

In order to prove $DDS_C(X, S') \subseteq DDS_C(X, S)$, several `m2o-lemmas` described above can be used.

In the other cases, systems are intentionally designed to have some constraints to avoid cyclic dependencies. For example, if a system is constrained to have no cyclic chains of links of objects, then there should be no cyclic dependency in the system no matter how the local states of the objects transit. Since the purpose of such constraints is to simplify complicated controls of dependencies of objects, it is typically easier to check the constraints than to use $noCycle_C$ defined above.

Definition [directly relating set]: The *directly relating set of object X in global state S w.r.t. relationship r*, denoted $DRS_r(X, S)$, is defined as $X' \in DRS_r(X, S)$ iff there is r between X and X' in S.

Definition [relating set]: Let X be an object, S a global state, and r a relationship of objects, then the *relating set of X in S w.r.t. r*, denoted $RS_r(X, S)$, is recursively defined as (1) $\forall X' : (r(X, X', S) \rightarrow X' \in RS_r(X, S))$, and (2) $\forall X', X'' : (X' \in RS_r(X, S) \wedge r(X', X'', S) \rightarrow X'' \in RS_r(X, S))$.

Definition [no cyclic relationship]: Let X be an object, S a global state, and r a relationship of objects, then we say *X is in no cyclic relationship in S w.r.t. r*, denoted $noCycle_r(X, S)$, iff X itself is not included in $RS_r(X, S)$.

Lemma: Let C is a class, X an object of C, S a global state and r a relationship of objects. If $DDS_c(X, S)$ is a subset of $DRS_R(X, S)$ for all X in S, then $noCycle_r(X, S)$ implies $noCycle_c(X, S)$ for all X.

This lemma allows the user of the framework to define `DDSC` implementing some simpler relationship r instead of the true DDS_C and use `noCycleC` defined by using the `DDSC` instead of the true $noCycle_c$.

6.3.4 Cyclic Dependency Lemma

When $noCycle_C(S)$ is an invariant, there may exist an applicable transition rule and the state machine continues to transit.

Cyclic Dependency Lemma: If there is no cyclic dependency of class C in global state S and there exists some object X of C in S whose local state is included in $prels(R, C)$, then there exists some object O of C in S such that the local state of O is included in $prels(R, C)$ and the depending set of O includes no object of C whose local state is included in $prels(R, C)$,.

For example, if there is no cyclic dependency of `Resource` in global state S which includes an initial resource, then there exists some initial resource in S whose depending set includes no initial resources.

7 Verification Procedure of Leads-to Properties

A typical property of an automated system setup operation, which we want to verify, is that the operation surely brings a cloud system to a global state where all of its resources are started. Let the automation of a setup operation be modeled as a state machine $TS = (St, Tr, In)$ specified by sort `State` and a set of transition rules, and let $Fn \subseteq St$ be a set of expected final states, reachability we want to verify is formalized as ($init$ `leads-to` $final$) where $init$ and $final$ are predicates for a given global state S such that $init(S)$ holds iff $S \in In$ and $final(S)$ holds iff $S \in Fn$.

In order to prove ($init$ `leads-to` $final$), we should find a state predicate p such that ($init \rightarrow p$) and (p `leads-to` $final$). However such p is specific to the individual problem, one of the most typical and general ones is that $p(S)$ means "S has a next state," i.e. "S will transit." When a state machine has such general p, it always continues to transit until it reaches a final state.

Definition [continuous predicate]: The *continuous predicate, cont*, is the predicate which holds iff there exists some next state of a given state.

Sufficient conditions: Let $TS = (St, Tr, In)$ be a state machine, inv a conjunction of some state predicates and m a natural number function of St, then the following five conditions are sufficient for ($init$ `leads-to` $final$) to hold.

$$\forall S \in St: \quad (init(S) \rightarrow cont(S)) \tag{1}$$
$$\forall (S, S') \in Tr: \quad ((inv(S) \wedge \neg final(S)) \rightarrow (cont(S') \vee final(S'))) \tag{2}$$
$$\forall (S, S') \in Tr: \quad ((inv(S) \wedge \neg final(S)) \rightarrow (m(S) > m(S'))) \tag{3}$$
$$\forall S \in St: \quad (init(S) \rightarrow inv(S)) \tag{4}$$
$$\forall (S, S') \in Tr: \quad (inv(S) \rightarrow inv(S')) \tag{5}$$

Condition (1) means an initial state should be a continuing state, i.e. it should start transitions. Condition (2) means transitions continue until $final(S')$ holds. Condition (3) implies that $m(S)$ keeps to decrease properly while $final(S)$ does not hold. Since $m(S)$ is a natural number, it should stop to decrease in finite steps and the state machine should get to state S' where $cont(S')$ does not hold and $final(S')$ does hold, because condition (2) ensures that transitions would still continue if $cont(S')$ does hold. Here, m is called a *state measuring function*. When condition (4) and (5) hold, each state predicate included in inv is called an invariant.

7.1 Procedure: Definition of Predicates

Step 0-1: Define $init$ and $final$.

Among conditions explicitly composing $init(S)$, one referring to no local states of objects and being expected to be an invariant is called a *wfs (well-formed state)* and we usually gather them

and define predicate wfs as a conjunction of them. Note that $wfs(S)$ becomes typically a long conjunction sequence which is not so efficient to evaluate. Fortunately, in the course of proving a leads-to property, wfs is used only in the antecedent part of condition (1) as a part of *init* and thus it is enough to know whether wfs does or does not reduce to false and Step 0-4 introduces a much efficient representation. Here, wfs is defined but is declared `nonexec` as CafeOBJ does not use the definition.

In addition, do not forget to include $noCycle_C$ in the *init* predicate when using the Cyclic Dependency Lemma.

Step 0-2: Define *cont*.

```
vars S SS : State
eq cont(S) = (S =(*,1)=>+ SS) .
```

Step 0-3: Define *m*.
We should find a natural number function that properly decreases in transitions. If we can model a cloud system as a state machine where every transition rule changes at least one local state of an object and there is no loop transition, then the measuring function, *m*, can be easily defined as the weighted sum of counting local states of all the classes of objects. Suppose that local states of class C are $st_C^0, st_C^1, \ldots, st_C^{n_c}$ and they are straightforward, that is, there is no backward transition, then m can be $\sum_C \sum_{0 \le k \le n_C} \#st_C^k \times (n_c - k)$ where $\#st_C^k$ is the number of objects of class C whose local state is st_C^k. When a rule makes an object of class C transit from state s_c^k to st_C^{k+1}, $\#st_C^k$ decreases by 1 and $\#st_C^{k+1}$ increases by 1 so that $m(S') = m(S) - (n_c - k) + (n_c - k - 1) = m(S) - 1$ holds.

```
var SetRS : SetOfResource
var SetPR : SetOfProperty
op m : State -> Nat
eq m(< SetRS,SetPR >)
   = (#ResourceInStates(initial,SetRS) * 1)
   + (#ResourceInStates(started,SetRS) * 0)
   + (#PropertyInStates(notready,SetPR) * 1)
   + (#PropertyInStates(ready,SetPR) * 0) .
```

When the state machine has a rule without changing any local states of objects, *m* should include an additional term that decreases when the rule is applied. But, instead, we recommend introducing some local state representing whether the rule is already applied or not yet.

When there is a loop transition, *m* should include an additional term that properly decreases whenever a loop occurs. The simplest approach is to introduce an object whose local state is a loop counter.

Step 0-4: Define *inv*.
Invariants other than wfs predicates are usually recognized to be necessary in the course of proving conditions (1) to (5) above and are introduced by the user of the framework.

Predicate *inv* is a conjunction of all the invariants including wfs predicates, however, the straightforward representation is not so efficient. Fortunately, there is more efficient representation. Since the sufficient conditions (2), (3), and (5) include *inv* in their antecedent parts, it is enough to know whether each invariant does or does not reduce to false. Thereby, we can define *inv* such that it reduces to false when one of invariants reduces to false as follows:

```
var S : State
pred inv : State
ceq wfs(S) = false if not wfs-AAAAA(S) .
ceq wfs(S) = false if not wfs-BBBBB(S) .
ceq inv(S) = false if not wfs(S) .
ceq inv(S) = false if not inv-CCCCC(S) .
ceq inv(S) = false if not inv-DDDDD(S) .
```

As to sufficient conditions (4) and (5), *inv* is also included in their consequent parts, which case will be explained below.

Step 0-5: Prepare for using the Cyclic Dependency Lemma.
When using the Cyclic Dependency Lemma, we firstly introduce an object which is in one of the pre-transit local states of a transition rule and then we claim that DDS_C of the object includes no object in the pre-transit local states.

CITP method used a `:init` command to introduce a lemma on the way of proofs. The lemma should be defined in the non-execute mode and be labeled in advance. The `:init` command is used to specify the labeled lemma and the appropriate substitution of variables. The following conditional equation is defined in advance and means that there is a contradiction when DDS_C of the specified resource includes any `initial` resource.

```
-- The Cyclic Dependency Lemma ensures
-- an initial resource whose DDSC includes no initial resourecs.
var T : RSType
var IDRS : RSID
var S : State
ceq [Cycle :nonexec]:
   true = false
   if someRSInStates(DDSC(res(T, IDRS, initial),S),initial) .
```

`Cycle` is the label of this lemma and `:nonexec` means that this lemma is executed only when it is introduced by a `:init` command and variables `T`, `IDRS`, and `S` are substituted. The usage of an `:init` command will be described below.

Step 0-6: Prepare arbitrary constants.
Proof scores for the sufficient conditions require many arbitrary constants, i.e. *proof constants*. In order to make the proof score be easy to understand, proof constants are consistently named and declared.

```
ops idRS idRS' idRS1 : -> RSIDLt
ops idRRS idRRS' idRRS1 : -> RSIDLt
ops idPR idPR' idPR1 : -> PRIDLt
ops sRS sRS' sRS'' sRS''' : -> SetOfResource
ops sPR sPR' sPR'' sPR''' : -> SetOfProperty
ops trs trs' trs'' trs''' : -> RSType
ops tpr tpr' tpr'' tpr''' : -> PRType
ops srs srs' srs'' srs''' : -> RSState
ops spr spr' spr'' spr''' : -> PRState
op stRS : -> SetOfRSState
op stPR : -> SetOfPRState
```

7.2 Procedure: Proof of Sufficient Condition (1)

The verification procedure is basically a process to repeat three actions; (1) pick up an unproved case which is then called the *current case*, (2) split the current case into cases which collectively cover the current case, (3) try to reduce the split cases to true.

Step 1-0: Define a predicate to be proved.
Predicate `initcont` to represent condition (1) can be defined as follows:

```
var S : State
pred initcont : State .
eq initcont(S) = init(S) implies cont(S) .
```

Step 1-1: Begin with the most general case.
In the most general case for proof of condition (1), the global state consists of proof constants every of which represents an arbitrary set of objects of each class.

```
:goal {eq initcont(< sRS, sPR >) = true .}
```

Step 1-2: Consider which rule can be applied to the global state in the current case. The rule is referred to as the *current rule*.

Step 1-3: Split the current case into cases which collectively cover the current case and one of which matches to LHS of the current rule.

Step 1-4: Split the current case into cases where the condition of the current rule does or does not hold.

Step 1-5: When there are two identifier constants, split the current case into cases where they are or are not the same.

Step 1-6: When there is a dangling link, split the current case into cases where the linked object does or does not exist.

Step 1-7: When falling in a cyclic situation, use the Cyclic Dependency Lemma.

```
:init [Cycle] by {
  T:RSType <- trs;
  IDRS:RSID <- idRS;
  S:State <- < sRS, sPR >;
}
```

7.3 Procedure: Proof of Sufficient Condition (2)

Step 2-0: Define a predicate to be proved.
Predicate `contcont` for condition (2) can be defined as follows:

```
vars S SS : State
var CC : Bool

pred ccont : State State
```

```
eq ccont(S,SS)
   = inv(S) and not final(S) implies cont(SS) or final(SS) .

pred contcont : State
eq contcont(S)
   = not (S =(*,1)=>+ SS if CC suchThat
          not ((CC implies ccont(S,SS)) == true)
          { true }) .
```

Step 2-1: Begin with the cases each of which matches to LHS of each rule.
Since condition (2) checks every possible next state of a given state *S*, we only need to prove the cases each of which matches to each rule.

```
-- Goal of Condition (2) for rule R01
:goal {
  eq contcont(< (res(trs,idRS,initial) sRS), sPR >) = true .
}
```

Step 2-2: Split the current case for a rule into cases where the condition of the rule does or does not hold.
Note that when the current rule is conditional, never try to reduce the current case before applying this step. In addition, when the condition of the current rule is complicated and more than one CITP commands are required to split the current case, do not try to reduce the split cases before they become detailed enough to include a case where the condition of the rule holds.

Step 2-3: Split the current case into cases where predicate $final$ does or does not hold in the next state.

Step 2-4: Consider which rule can be applied to the next state. The rule is referred to as the *current rule*.

Step 2-5: Split the current case into cases which collectively cover the current case and the next state of one of the split cases matches to LHS of the current rule.

Step 2-6: Split the current case into cases where the condition of the current rule does or does not hold in the next state.

Step 2-7: When there are two identifier constants, split the current case into cases where they are or are not the same.

Step 2-8: When there is a dangling link, split the current case into cases where the linked object does or does not exist.

Step 2-9: When falling in a cyclic situation, use the Cyclic Dependency Lemma.

7.4 Procedure: Proof of Sufficient Condition (3)

Since the antecedent part of condition (3) is equivalent to (2), the proof procedure of (3) is almost the same as of (2).

Step 3-0: Use natural number axioms.
Since the standard sort `Nat` of CafeOBJ does not have enough information to deduce natural number expressions, the framework provides module `NATAXIOM` which includes several axioms to be used for proof of condition (3).

```
protecting(NATAXIOM)
```

Step 3-1: Define a predicate to be proved.
Predicate `mesmes` for condition (3) can be defined as follows:

```
vars S SS : State
var CC : Bool

pred mmes : State State .
eq mmes(S,SS)
   = inv(S) and not final(S) implies m(S) > m(SS) .

pred mesmes : State .
eq mesmes(S)
   = not (S =(*,1)=>+ SS if CC suchThat
          not ((CC implies mmes(S,SS)) == true)
          { true }) .
```

Step 3-2: Begin with the cases each of which matches to LHS of each rule.

```
-- Goal of Condition (3) for rule R01
:goal {
  eq mesmes(< (res(trs,idRS,initial) sRS), sPR >) = true .
}
```

Step 3-3: Split the current case for a rule into cases where the condition of the rule does or does not hold.
Note that when the current rule is conditional, never try to reduce the current case before applying this step. In addition, when the condition of the current rule is complicated and more than one CITP commands are required to split the current case, do not try to reduce the split cases before they become detailed enough to include a case where the condition of the rule holds.

Step 3-4: When there are two identifier constants, split the current case into cases where they are or are not the same.

Step 3-5: When there is a dangling link, split the current case into cases where the linked object does or does not exist.

7.5 Procedure: Proof of Sufficient Condition (4) & (5)

Since predicate *inv* typically is a conjunction of many predicates, it is better to prove each of them separately. Suppose $inv(S) = inv_1(S) \wedge inv_2(S) \wedge \cdots \wedge inv_n(S)$, then we can separately prove the invariant property of each $inv_k(S)$ since the followings hold:

$$\forall S \in St : (\forall k : init(S) \rightarrow inv_k(S)) \rightarrow (init(S) \rightarrow inv(S))$$

$$\forall (S, S') \in Tr : (\forall k : inv(S) \rightarrow inv_k(S')) \rightarrow (inv(S) \rightarrow inv(S'))$$

Step 4-0: Define a predicate to be proved.
Predicate `initinv` for condition (4) can be defined as follows:

```
vars S : State

pred invK : State
pred initinv : State
eq initinv(S) = init(S) implies invK(S) .

eq invK(S) = inv-AAAAA(S) .
```

Step 4-1: Begin with the most general case.

```
:goal {
  eq initinv(< sRS,sPR >) = true .
}
```

Step 5-0: Define a predicate to be proved.
Predicate `invinv` for condition (5) can be defined as follows:

```
vars S SS : State
var CC : Bool

pred iinv : State State .
eq iinv(S,SS) = inv(S) and invK(S) implies invK(SS) .

pred invinv : State
eq invinv(S)
   = not (S =(*,1)=>+ SS if CC suchThat
          not ((CC implies iinv(S,SS)) == true)
          { true }) .

eq invK(S) = inv-AAAAA(S) .
```

However $inv(S)$ includes $inv_k(S)$, the antecedent part of `iinv(S)` doubly specifies `invK(S)`. This is because `inv(S)` is defined only to reduce to false when one of `invK(S)` reduces to false

Step 5-1: Begin with the cases each of which matches to LHS of each rule.

```
-- Goal of Condition (5) for rule R01
:goal {
  eq invinv(< (res(trs,idRS,initial) sRS), sPR >) = true .
}
```

Step 5-2: Split the current case for a rule into cases where the condition of the rule does or does not hold.
Note that when the current rule is conditional, never try to reduce the current case before applying this step. In addition, when the condition of the current rule is complicated and more than one CITP commands are required to split the current case, do not try to reduce the split cases

before they become detailed enough to include a case where the condition of the rule holds.

Step 5-3: When there are two identifier constants, split the current case into cases where they are or are not the same.

Step 5-4: When there is a dangling link, split the current case into cases where the linked object does or does not exist.

Step 5-5: Find a predicate which should keep to hold at the current and next states and consider which lemma ensures it. The lemma also should be proved unless it is proved and provide by the framework.

Proof of $noCycle_C$ as an Invariant

We should prove the invariant property of $noCycle_C(S)$ in order to use the Cyclic Dependency Lemma, however, it is included in *init* and thus we only need to prove condition (5) for $noCycle_C$. The Depending Subset Lemma ensures that we should prove that $\forall (S, S') \in Tr, \forall X \in C : DDS_C(X, S') \subseteq DDS_C(X, S)$ instead of condition (5).

Step 5-0: Define a predicate to be proved.

```
vars S SS : State
var CC : Bool
var RS : Resource
-- When subset(DDSC(RS,SS),DDSC(RS,S)) holds for all RS,
-- noRSCycle(S) implies noRSCycle(SS),
pred invnoRSCycle : Resource State
eq invnoRSCycle(RS,S)
   = not (S =(*,1)=>+ SS if CC suchThat
          not ((CC implies subset(DDSC(RS,SS),DDSC(RS,S))) == true)
          { true }) .
```

Step 5-1: Begin with the cases each of which matches to LHS of each rule.

Step 5-2: Split the current case for a rule into cases where the condition of the rule does or does not hold.
Note that when the current rule is conditional, never try to reduce the current case before applying this step. In addition, when the condition of the current rule is complicated and more than one CITP commands are required to split the current case, do not try to reduce the split cases before they become detailed enough to include a case where the condition of the rule holds.

7.6 Recommended Module Structure

The framework provides a recommended module structure which the user can adopt when developing proof scores for verifying the property (*init* `leads-to` *final*). Using the recommended structure results in proof scores which are consistent and easier to understand. Figure 7.6 depicts the recommended module structure whereas each box represents a module and each dashed arrow represents a "protecting" or "extending" import of another module. An italic name means a template module.

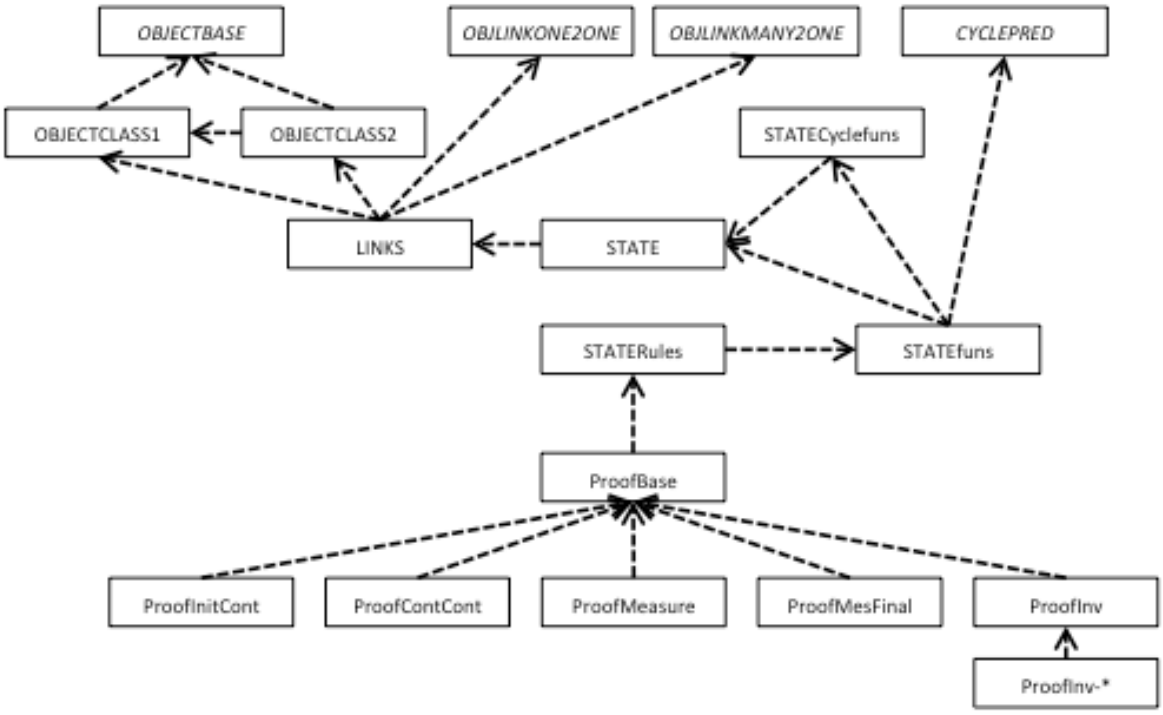

Figure 2: Recommended Module Structure

The following list describes the role and content of each module:

- `OBJECTCLASS`$_n$
  Module for each class of objects. This class should be named as representing the class appropriately. The name usually consists of upper case letters because the same name will be capitalized and used for the sort of the class. The contents of this module is as follows:

  1. Protecting import the modules of other classes which this class links.
  2. Extending import the template module `OBJECTBASE` and rename predefined sorts and operators for the class.
  3. Define the constructor of the class.
  4. Define literals of the type and local state of the class.
  5. Define the selectors of the class.
  6. Define operators that are specific to the class if any.

- `LINKS`
  Module for links between objects.

  1. Protecting import the modules of classes of links.
  2. Extending import the template modules `OBJLINKONE2ONE` and `OBJLINKMANY2ONE`, and rename predefined sorts and operators for links between objects.

- `STATE`
  Module for global states.

1. Protecting import LINKS.
2. Define sort State for representing global states. A global state is usually represented as a tuple of sets of objects, each of the sets is a finite subset of a class.

- STATECyclefuns
  Module for preparing to use the Cyclic Dependency Lemma.

  1. Protecting import STATE.
  2. Define operator getAllObjInState.
  3. Define operator DDSC.

- STATEfuns
  Module for defining many kinds of operators for global states.

  1. Protecting import STATE.
  2. Extending import the template module CYCLEPRED with STATECyclefuns as a parameter module, and rename predefined operator noCycleC.
  3. Define wfs predicates.
  4. Define predicates init, final, and wfs.
  5. Define operators required to implement standard predicates above if any.

- STATERules
  Module for transition rules.

  1. Protecting import STATEfuns.
  2. Define transition rules.

- ProofBase
  Module for common definitions to prove five sufficient conditions.

  1. Protecting import STATERules.
  2. Define invariant predicates.
  3. Define predicate cont.
  4. Define operator m.
  5. Define predicate inv such that it reduces to false when one of invariants reduces to false.
  6. Define problem specific lemmas if any.
  7. Prepare proof constants.

- ProofInitCont
  Module for proving sufficient condition (1).

  1. Protecting import ProofBase.
  2. Define predicate initcont.
  3. Define problem specific lemmas if any.

- `ProofContCont`
  Module for proving sufficient condition (2).

  1. Protecting import `ProofBase`.
  2. Define predicate `contcont`.
  3. Define problem specific lemmas if any.

- `ProofMeasure`
  Module for proving sufficient condition (3).

  1. Protecting import `ProofBase`.
  2. Define predicate `mesmes`.
  3. Define axioms of `Nat`.
  4. Define problem specific lemmas if any.

- `ProofInv`
  Module for common definitions for proving sufficient condition (4) and (5).

  1. Protecting import `ProofBase`.
  2. Define predicates `invK`, `initinv`, and `invinv`.

- `Proofinv-* Proofwfs-*`
  Module for proving each invariant.
  The name of this module is usually `Proof+`*name_of_invariant*.

  1. Protecting import `ProofInv`.
  2. Define predicates `invK` to be the invariant.
  3. Define lemmas if necessary.