An Interactive Theorem Proving Framework for Declarative Cloud Orchestration

by

Hiroyuki Yoshida

submitted to
Japan Advanced Institute of Science and Technology
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Supervisor: Professor Kokichi Futatsugi

School of Information Science
Japan Advanced Institute of Science and Technology

December 03, 2016

Abstract

An interactive theorem proving framework for verifying liveness properties of declarative cloud orchestration is proposed.

Recent rapid progress of cloud computing accelerates the whole life cycle of system usage and requires much flexible automation of system operations. Automation of cloud system operations is called cloud orchestration and correctness of cloud orchestration becomes much crucial for many activities in the human society. However, correctness of automated system operations cannot depend on testing-based quality control because tests of them are meaningful intrinsically when done on production environments not on testing environments. Formal approaches are expected to provide systematic ways to guarantee correctness of cloud orchestration.

Formal approaches are mainly classified into two categories, model checking and theorem proving. As opposed to model checking, theorem proving can verify models of arbitrary many number of states and so suitable for proving absence of counter examples. However, when applying to practical problems it requires many human efforts to develop proofs.

This paper proposes a framework of interactive proof development for a kind of liveness properties, leads-to property, of cloud orchestration. We say "framework" to mean something like an application framework of software development which brings high productivity by minimizing development efforts and high maintainability by consistent structure of application software.

The proposed framework provides (1) a general way to formalize specifications of different kinds of cloud orchestration tools and (2) a procedure for how to verifying a kind of liveness properties, as well as invariant properties, of formalized specifications. It also provides (3) general templates and libraries of formal descriptions for specifying orchestration of cloud systems and (4) proved lemmas for general predicates of the libraries to be used for verification.

The framework has been applied to the verification of specifications of AWS CloudFormation and also of OASIS TOSCA, and is demonstrated to be effective for reducing generic routine work and making a verification engineer concentrate on the work specific to each individual system. The example of OASIS TOSCA shows that the framework can be used to specify, represent, and verify the behavior models of TOSCA where the standard has not yet provided any ways to do so. It also shows a general way to manage dependencies of cloud resources which is a smarter one than that of the most popular tool, CloudFormation.

The major contributions of this paper are (1) it shows that cloud orchestration is a practical and suitable domain to apply interactive theorem proving and (2) it introduces the idea of frameworks from software development to proof development which results in high productivity and high maintainability of proofs.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my supervisor, Professor Kokichi Futatsugi for his acute guidance, encouragement, and unlimited support throughout the duration of my research. Without his support, this thesis could not have been completed. In addition, I am indebted to him for giving me this invaluable opportunity to study abroad and providing me the financial support.

I would like to thank Professor Kazuhiro Ogata for his feedback and wise advices on my research. The quality of this research was significantly improved because of him.

My last acknowledgements go to my family for their support, unconditional love and vital encouragement throughout my study and throughout my life.

Contents

List of Figures

Chapter 1

Introduction

Cloud computing has recently emerged as an important infrastructure supporting many aspects of human activities. In former days, it took several months to make system infrastructure resources (computer, network, storage, etc.) available, while in these days, it takes only several minutes to do so. This situation accelerates the whole life cycle of system usage where much flexible automation is required for system operations such as setting up, scaling, patching, and so on. The total automation of system operations is sometimes called "Infrastructure as Code" (IaC).

Correctness of automated operations of cloud systems is much more crucial than that of the former systems because cloud systems serve to much more people in much longer time than the former systems used mainly inside of companies. Cloud computing enables to easily, cheaply, and repeatedly prepare testing environments for applications, however, tests of system operations are meaningful intrinsically when done on production environments not on testing environments. It means that test-based quality control is not sufficient for automated system operations.

A system on cloud consists of many "parts," such as virtual machines (VMs), storages, and network services as well as software packages, configuration files, and user accounts in VMs. These parts are called *resources* and the automated management of cloud resources is called *resource orchestration*, or *cloud orchestration*.

The most popular cloud orchestration tool is *CloudFormation* [1] provided as a service by Amazon Web Services (AWS) and a compatible open source tool is being developed as *OpenStack Heat* [21]. CloudFormation can manage resources provided by IaaS platform of AWS, such as VMs (EC2), block storages (EBS), load balancers (ELB), and so on. However, CloudFormation does not directly manage resources inside VMs and instead it allows to specify any types of scripts for initially setting up VMs, such as installing Httpd package, creating configuration files, copying contents, and activating an Httpd component. Shell command scripts were commonly used for this layer of management and recently several open source tools become popular such as *Puppet* [17], *Chef* [4], and *Ansible* [18]. People have to learn and use these several kinds of tools in actual situations, which results in much elaboration to guarantee its correctness. In an actual commercial experience of the author, more than 50% of troubles are caused by defects in those dependency definitions and scripts.

While orchestration tools are specialized into two management layers, on IaaS and inside VMs, there is a unified standard specification language, *OASIS TOSCA* [14] that can be used to describe the structure of both types of resources. The resource structure is called a *topology* and a TOSCA tool is expected to automate system operations based on resource dependencies

declaratively defined by topologies. Currently, however, there is no practical implementation of declarative specifications of TOSCA because it has not yet explicitly provided any ways to specify behavior of a topology, i.e. how to automate a topology.

We believe formal approaches will provide systematic ways to guarantee correctness of cloud orchestration. Formal approaches are mainly classified into two categories, *model checking* and *theorem proving*. Model checking methods are based on exhaustive analysis of states of transition systems and can automatically find counter examples included in the specified models. However, the sizes of models are limited and thus absence of counter examples can not be proved. On the other hand, theorem proving can verify models of arbitrary many number of states and so suitable for proving absence of counter examples. It requires thinking through meanings of the specified models, which is very important aspect of developing trusted systems. However, when applying to practical problems, it requires many human efforts to develop proofs for splitting the cases, establishing lemmas, and proving them in the course of verification.

This paper proposes a framework of interactive proof development for a kind of liveness properties, *leads-to* property, of cloud orchestration. Here we say "framework" to mean something like an application framework of software development. For example, Ruby on Rails (RoR) [10] is one of the most popular application frameworks. RoR defines an MVC architecture of web applications, provides super classes and utility classes to implement the architecture, and gives developers a guide for how to design and code web applications. Focusing on a specific application domain, namely web applications, RoR brings high productivity by minimizing development efforts and high maintainability by consistent structure of applications.

Similarly, our framework provides a general formalization of cloud orchestration specifications of different kinds of tools and provides a procedure for how to verify leads-to properties, and also invariant properties, of the specifications. It also provides logic templates and predicate libraries which are defined in a general level of abstraction and can be instantiated as problem specific descriptions, predicates, and lemmas. Using them, the verification procedure assists developers to systematically think and develop proofs of leads-to properties.

The rest of this paper is organized as follows. Chapter 2 introduces several cloud orchestration tools. Chapter 3 introduces functionalities of CafeOBJ language in which we represent formal specifications of cloud systems. Chapter 4 describes a general model of cloud orchestration. Chapter 5 describes general logic templates and predicate libraries. Chapter 6 presents the procedure for verification of leads-to properties using a simple example specification of CloudFormation. Chapter 7 explains how the framework is applied to verification of OASIS TOSCA specifications. Chapter 8 explains related work and future issues.

Chapter 2

Cloud Orchestration

2.1 AWS CloudFormation

The most popular cloud orchestration tool is *CloudFormation* [1] provided as a service by Amazon Web Services (AWS). CloudFormation can manage *resources* provided by IaaS platform of AWS, such as VMs (EC2), block storages (EBS), and load balancers (ELB). CloudFormation automatically sets up these resources according to a *template* that declaratively defines dependencies of resources. A template is a set of resources and a resource has an identifier and a type and includes several *properties* which may depend on other resources.

Fig. 2.1 is part of a very simple CloudFormation template written in JSON format [11]. This template specifies the dependency of the resources in a simple cloud system on the AWS IaaS platform shown in lower part of Fig. 2.1. There are two resources; one has the identifier `MyInstance` and the type `AWS::EC2::Instance`, another has the identifier `MyEIP` and the type `AWS::EC2::EIP`. Note that an Elastic Compute Cloud instance (EC2 instance) is a virtual machine on AWS IaaS platform and an Elastic IP (EIP) provides a static IP address for an EC2 instance which is dynamically created and activated. `MyEIP` has a property whose type is `InstanceID`. The property refers `MyInstance` which makes resource `MyEIP` depend on `MyInstance`. Thus, CloudFormation firstly activates `MyInstance` and then activates `MyEIP` with a parameter of the instance ID of `MyInstance`.

```
{ "Resources" : {
    "MyInstance" : {
      "Type" : "AWS::EC2::Instance",
    "MyEIP" : {
      "Type" : "AWS::EC2::EIP",
      "Properties" : {
        "InstanceId" : { "Ref" : "MyInstance" }
}}}}}
```

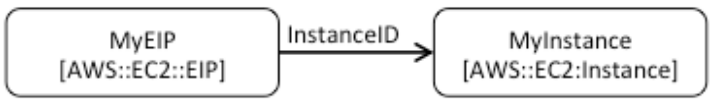

Figure 2.1: A Very Simple CloudFormation Template

```
package { "httpd":
  name => "httpd",
  ensure => "installed"
}
service { "httpd":
  name => "httpd",
  enable => "true",
  ensure => running,
  require => Package["httpd"]
}
file { "/var/www/html/sample":
  ensure => directory,
  owner => "apache",
  group => "apache",
  mode => "755",
  require => Package["httpd"]
}
file { "/var/www/html/sample/sample.html":
  source => "puppet:///files/sample.html",
  owner => "root",
  group => "root",
  mode => "644",
  require => File["/var/www/html/sample"]
}
file { "/etc/httpd/conf/httpd.conf":
  source => "puppet:///files/httpd.conf",
  mode => "644",
  owner => "root",
  group => "root"
  subscribe => Service["httpd"]
}
```

Figure 2.2: A Simple Puppet Manifest for Setting up an HTTPD Server

2.2 Puppet, Chef, and Ansible

Puppet, Chef, and Ansible are not cloud orchestration tools but *deployment tools* (also called *configuration tools*) which are used by cloud orchestration tools to set up resources inside VMs.

Puppet [17] provides a domain specific language (DSL) to describe executable Ruby [12] scripts for setting up resources inside VMs. A Ruby script of Puppet is called a *manifest* and executed in a VM. Fig. 2.2 is a simple manifest written in Puppet DSL to set up an httpd service. A manifest is a list of declarations each of which declares the desired state of a resource. Each declaration specifies a type, a title, and attributes of a resource. For example, the first four lines of the manifest shown in Fig. 2.2 specifies that a *package* type resource named `httpd` is desired to be installed to the target VM. A package means an installable package of a middleware which is the Apache HTTP server in this case. The second declaration of the example manifest means that an httpd service is desired to be running and it requires the httpd package resource above is in the specified state. The third one means that the specified directory is desired to exist. The fourth means that a file is desired to be copied from the Puppet server to the specified file path

```
package "httpd" do
  action :install
end
cookbook_file "/etc/httpd/conf/httpd.conf" do
  source "httpd.conf"
  owner 'root'
  group 'root'
  mode 00644
end
directory "/var/www/html/sample" do
  owner 'apache'
  group 'apache'
  mode 00755
  action :create
end
cookbook_file "/var/www/html/sample/sample.html" do
  source "sample.html"
  owner 'root'
  group 'root'
  mode 00644
end
service "httpd" do
  supports :status => true, :restart => true, :reload => true
  action [:enable,:start]
end
```

Figure 2.3: A Simple Chef Recipe for Setting up an HTTPD Server

whose owner is `root`, group is `root`, and mode is 644, which requires the directory declared above is in the specified state. The fifth one also means that a file is desired to be copied and this resource should be checked whenever the state of the specified service resource is changed. A manifest is not necessarily executed from top to bottom; the order is decided by `require` and `subscribe` attributes. A manifest is idempotent which means the result of its execution is always the same because nothing is done when a specified resource is already in the desired state.

Chef [4] also provides a Ruby-based DSL to describe executable scripts for setting up resources inside VMs. A script is called a *recipe* and executed in a VM. A collection of related recipes and auxiliary files is called a *cookbook*. Fig. 2.3 is a simple recipe written in Chef DSL to set up an httpd service. A recipe is a list of declarations each of which declares the desired state of a resource. Each declaration specifies a type, a name, attributes, and actions of a resource. For example, the first three lines of the recipe shown in Fig. 2.3 specifies that a package type resource named `httpd` is desired to be installed to the target VM and the action to do so is `:install`. The second declaration of the example recipe means that a file `httpd.conf` included in the cookbook of this recipe is desired to be copied to the specified file path whose owner is `root`, group is `root`, and mode is 00644. The third one means that the specified directory is desired to exist and the fifth one means that an httpd service is desired to be running. As opposed to a Puppet manifest, a Chef recipe is executed from top to bottom and so the order of resources is critical; the fourth and fifth resources should not be inverted because the directory should exist before the file is copied into it. A recipe is idempotent similarly as a

```
A: one
B:
  - C: two
    D: three
  - E: four
    F: five
G: six
```

Figure 2.4: An Example YMAL Document

Puppet manifest. Since actions to achieve the desired states (e.g. `:install`) are abstracted and implemented for many kinds of operating systems, a recipe is independent from the difference of them.

What corresponds to a manifest of Puppet or a recipe of Chef is called a *playbook* in Ansible [18]. Although Ansible is implemented by Python [22], a playbook is not an executable script in Python but a YAML format file [5] which is interpreted and executed by `ansible-playbook` command. Before showing an example playbook, we will briefly explain the YAML format. A YAML document represents nesting key-value lists and arrays. A key and its value are separated by a colon (:). Keys with the same indentation composes a list. In Fig. 2.4, the top level list has three key-value pairs whose keys are `A`, `B`, and `G`. An array is represented by minus signs (-) with the same indentation. In the figure, the value of key `B` is an array with two elements each of which is a key-value list with two pairs. Let us write a key-value list as `{(k1, v1), (k2, v2), ... }`and an array as `[e1, e2, ... ]`, then the data structure represented by the YAML document in Fig. 2.4 is the following list:

```
{(A, one), (B, [{(C, two), (D, three)}, {(E, four), (F, five)}]), (G, six)}
```

A playbook represents an array of *plays* which are a sports analogy; many plays are required to set up a cloud system. Fig. 2.5 is a simple example playbook to set up an httpd service which represents only one play. A play is a key-value list including keys of `hosts`, `tasks`, and so on. Key `hosts` specifies machines to which the play is applied. Key `tasks` specifies an array of *tasks* which are executed in order. In the example, the task array includes five tasks. A task is a key-value list and typically begins with the pair of `name` key and its value which serves as a comment. The second pair of a task specifies a *module* and the parameters to invoke it. A module is a command provided by Ansible which can be remotely executed on the specified VMs. In the example, module `yum` will install the package resource named `httpd`, module `file` will create the specified file or directory, and `service` will start the httpd service to be running. Similarly as a Chef recipe, tasks in an Ansible playbook are executed from top to bottom and so the order of tasks is critical. A playbook is idempotent similarly as a Puppet manifest and a Chef recipe. Since modules to achieve the desired states are abstracted and implemented for many kinds of operating systems, a playbook is independent from the difference of them.

Although there are several differences among Puppet, Chef, and Ansible which are omitted to explain here, they share several common features in comparison with shell command scripts provided by operating systems of VMs. They provide domain specific languages to describe the desired states of resources. The descriptions in the DSLs are idempotent and abstracted to be independent from the difference of operation systems.

However, people have to learn and use at least two different kinds of tools (orchestration tools and configuration tools) with different styles of specifications and functionalities, which results in much elaboration to guarantee the correctness of automated system operations.

```
- hosts: webservers
  tasks:
    - name: be sure httpd is installed
      yum: name=httpd state=installed
    - name: be sure httpd.conf exists
      file: src=/file/httpd.conf
            path=/etc/httpd/conf/httpd.conf
            state=file
            owner=root
            group=root
            mode=0644
    - name: be sure springboot root directory exists
      file: path=/var/www/html/sample
            state=directory
            owner=apache
            group=apache
            mode=0755
    - name: be sure sample.html exists
      file: src=/file/sample.html
            path=/var/www/html/sample/sample.html
            state=file
            owner=root
            group=root
            mode=0644
    - name: be sure httpd is running and enabled
      service: name=httpd state=running enabled=yes
```

Figure 2.5: A Simple Ansible Playbook for Setting up an HTTPD Server

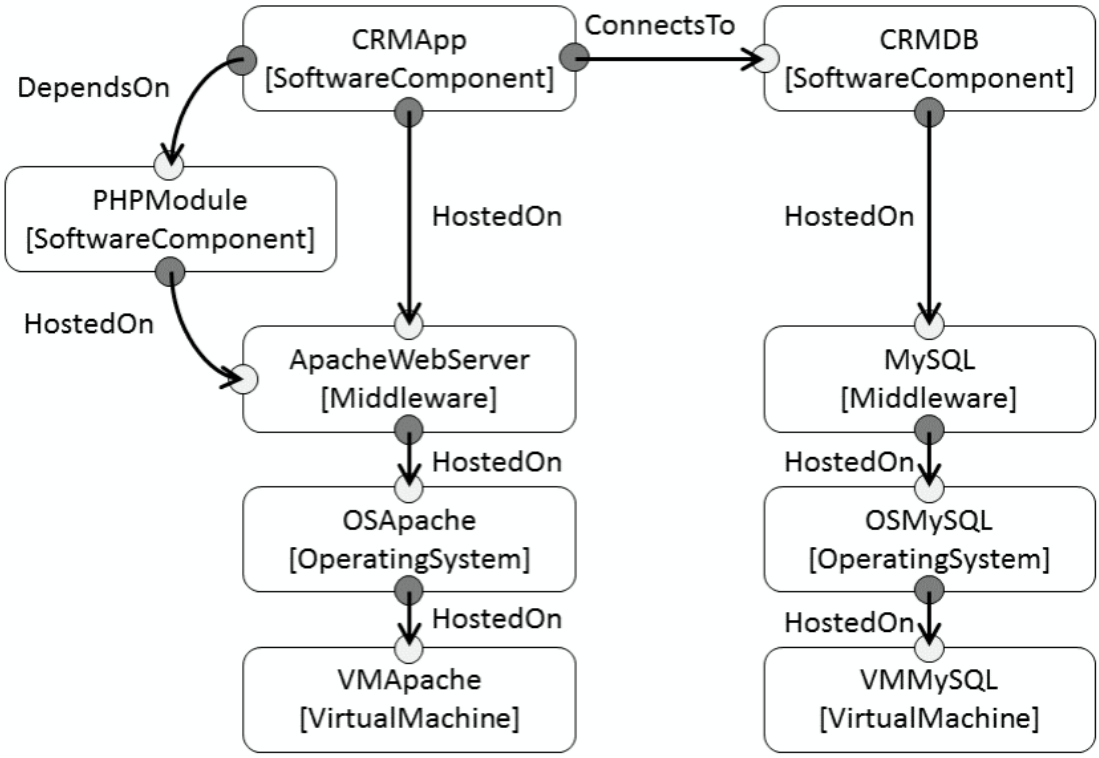

Figure 2.6: An Example of TOSCA topology

2.3 OASIS TOSCA

OASIS TOSCA[14] is a standard specification language to describe automation of a cloud system consisting of service components and their relationships using a *service template*. It provides interoperable deployments of cloud systems across different cloud environments and their management throughout the complete lifecycle (e.g. setting up, scaling, patching, monitoring, etc.). A service template consists of a *topology template* and optionally a set of *plans*. A topology template defines the resource structure of a cloud system. Note that a topology template can be parameterized to give actual environment parameters such as IP addresses, which is the reason why named as "template" and in this paper we simply say "a topology" for the sake of brevity. A plan is an imperative definition of a system operation of the cloud system, such as a setup plan, written by a standard process modeling language, such as BPMN [16].

In TOSCA, a resource is called a *node* that has several *capabilities* and *requirements*. A topology consists of a set of nodes and a set of *relationships* of nodes. A capability is a function that the node provides to another node, while a requirement is a function that the node needs to be provided by another node. A relationship relates a requirement of a source node to a capability of a target node. Note that nodes and relationships in a topology template can also be parameterized, thus the exact terms of TOSCA are node templates and relationship templates. Fig. 2.6 shows a typical example of topology that consists of nine nodes and nine relationships. White circles represent capabilities and black ones are requirements.

The current version of TOSCA is an XML-based language[1]. Fig 2.7 is part of the topology template of Fig. 2.6. In this example, there are two nodes (`VMApache` and `OSApache`) and one relationship. `VMApacheOS` is a capability of `VMApache` and `OSApacheContainer` is a

[1]OASIS TOSCA TC has published the committee draft of a simple profile for a YAML-based language. [15]

```
<TopologyTemplate>
  <NodeTemplate id="VMApache" name="VM for Apache"
                type="VirtualMachine">
    <Capabilities>
      <Capability id="VMApacheOS" name="OS"
                  type="OperatingSystemContainerCapability"/>
    </Capabilities> </NodeTemplate>
  <NodeTemplate id="OSApache" name="OS for Apache"
                type="OperatingSystem">
    <Requirements>
      <Requirement id="OSApacheContainer" name="Container"
                   type="OperatingSystemContainerRequirement"/>
    </Requirements>
    <Capabilities>
      <Capability id="OsApacheSoftware" name="Software"
                  type="SoftwareContainerCapability"/>
    </Capabilities> </NodeTemplate>
  <RelationshipTemplate id="OSApacheHostedOnVMApache"
                        name="hosted on" type="HostedOn">
    <SourceElement ref="OSApacheContainer"/>
    <TargetElement ref="VMApacheOS"/>
  </RelationshipTemplate>
...
</TopologyTemplate>
```

Figure 2.7: A Topology Template of TOSCA

requirement of `OSApache`. Each node, relationship, capability, and requirement has a *type*, such as `VirtualMachine`, `HostedOn`, and so on. Types are main functionalities of TOSCA that enable reusability of topology descriptions.

TOSCA assumes two main engineering roles, namely a type architect and an application architect. In a typical scenario, type architects define and provide several types of those elements and an application architect uses them to define a topology of a cloud system. The type architect also defines *operations*[2] of node types, such as creating, starting, stopping, or deleting nodes, and of relationship types, such as attaching relationships. A system operation of a cloud system is implemented as an invocation sequence of the type operations, which can be decided in two kinds of manners. One is an imperative manner in which the application architect uses a process modeling language to define a plan that explicitly invokes these type operations. Another is a declarative one in which the application architect only defines a topology and a TOSCA tool will automatically invoke appropriate type operations based on the defined topology. Naturally, the declarative manner is a main target of OASIS TOSCA because it promotes more abstract and reusable descriptions of topologies.

In this paper, *behavior of topologies* means when and which type operations should be invoked in automation. It is important to notice that behavior of a topology is decided by types of included nodes and relationships. We also say *behavior of a type* to mean that the conditions and results of invoking its type operations, which is defined by a type architect. Usually, different types of nodes are provided by different vendors and so specified by different type architects. An application architect is responsible for behavior of a topology whereas type architects are responsible for behavior of their defined types.

Currently there are no practical implementations of the declarative manner of TOSCA and one of the reasons is that no standard set of type operations of nodes or relationships are defined and there is no way for type architects to define behavior of their own types. In Section 7.2, we will describe how to use our framework to define behavior of TOSCA types and to verify that a specified topology can correctly automate to set up the cloud system.

[2]In this paper, we say *a type operation* as an operation of a type whereas TOSCA calls it *a lifecycle operation*.

Chapter 3

Preliminaries of CafeOBJ

CafeOBJ [2] is a formal specification language that is one of the state-of-the-art algebraic specification languages and a member of the OBJ [9] language family, such as Maude [13]. CafeOBJ specifications are executable by regarding equations and transition rules in them as left-to-right rewrite rules, and this executability can be used for interactive theorem proving.

3.1 Modules and Equations

Basic units of specifications in CafeOBJ are *modules*. A module[1] consists of declarations of *module importations, sorts, sub-sort relations, operators, variables, equations* and *transition rules*, some of which may be omitted. Conventionally, names of modules, sorts, and variables are capitalized while names of operators including constants start with lower case letters or use punctuation symbols.

Modules may have *parameters* and are called parameterized modules if so. An example of parameterized modules is as follows [2]:

```
module! SET(X :: TRIV) {
  -- Module Importation
  protecting(NAT)

  -- Sorts, Sub-sort Relations
  [Elt.X < Set]

  -- Operators
  op empty : -> Set {constr}
  op _ _ : Set Set -> Set {constr assoc comm idem id: empty}

  op #_ : Set -> Nat
  op _U_ : Set Set -> Set
  pred _\in_ : Elt.X Set
  op _A_ : Set Set -> Set
  op _\\_ : Set Set -> Set
```

[1] CafeOBJ modules can be classified into tight modules and loose modules. Roughly speaking, a tight module denotes a unique model, while a loose module denotes a class of modules. Those are declared with `module!` and `module*` respectively.

[2] In CafeOBJ , a comment starts with `--` or `**` to the end of the line.

```
  pred subset : Set Set

  -- Variables
  vars S S1 S2 : Set
  vars E E1 : Elt.X

  -- Equations
  -- for =
  eq ((E S1) = (E S2)) = (S1 = S2) .
  -- for empty
  eq ((E S) = empty) = false .
  -- for #_
  eq # empty = 0 .
  eq # (E S) = 1 + (# S) .
  -- for _U_
  eq S1 U S2 = S1 S2 .
  -- for _\in_
  eq E \in empty = false .
  eq E \in (E S) = true .
  ceq E \in (E1 S) = E \in S if not(E = E1) .
  -- for _A_
  eq empty A S2 = empty .
  eq (E S1) A (E S2) = E (S1 A S2) .
  ceq (E S1) A S2 = S1 A S2 if not(E \in S2) .
  -- for _\\_
  eq empty \\ E = empty .
  eq (E S) \\ E = S .
  ceq (E1 S) \\ E = (E1 (S \\ E)) if not (E = E1) .
  -- for subset
  eq subset(empty,S) = true .
  eq subset((E S1),S2) = E \in S2 and subset(S1,S2) .
}
```

This module specifies generic sets and has one parameter `X` constrained by the built-in module `TRIV` in which one sort `Elt` is only declared as follows:

```
module* TRIV {
  [Elt]
}
```

The sort is referred by `Elt.X` and used for elements in `SET`. The built-in module `NAT` in which natural numbers are specified is imported with `protecting`. Modules also can be imported with `extending` and `using`; `protecting` means that elements of the imported modules should not be added nor collapsed; `extending` means that they can only be added but not be collapsed; and `using` means they can be added and collapsed.

One sort `Set` is declared and it is also declared that `Elt.X` is a sub-sort of `Set`. This is why an element is also a singleton set that only consists of the element. Operators may be constructors and a constructor without arguments is a constant. The operator `empty` is a constant of `Set` and the juxtaposition operator `_ _` is a constructor of `Set`, where an underscore is the place where an argument is put. It is also specified that the juxtaposition operator is associative, commutative, and idempotent and has `empty` as its identity. Operators are defined with

equations. The first equation specifies that `# empty` equals `0`, and the second one specifies that `# (E S)` equals `1 + (# S)`. Those two equations define operator `#_` that counts the number of the elements in a given set. Operators `_U_`, `_\in_`, `_A_`, `_\\_`, and `subset` are defined which mean union($\cup$), membership($\in$), intersection($\cap$), difference($\setminus$), and inclusion($\subseteq$) of sets respectively. Note that "`pred Op : Sort1 Sort2`" is an abbreviation for "`op Op : Sort1 Sort2 -> Bool`."

Parameterized modules can be instantiated with modules as actual parameters through views. Let us consider the following module as an actual parameter of `Set`:

```
module! SERVICE {
 protecting(NAT)
 [LocalState Service]
 ops closed open ready : -> LocalState {constr}
 op sv : Nat LocalState -> Service {constr}
}
```

in which two sorts are declared. A term of sort `LocalState` represents a local state of a service and there are three constants of local states (`closed`, `open`, and `ready`). A term of sort `Service` represents a service which has a form `sv(n,lst)` where `n` is some natural number as an identifier and `lst` is one of local states. `SET` can be instantiated as `SV-SET` as follows:

```
module! SV-SET {
 protecting(
  SET(SERVICE{sort Elt -> Service})
   * {sort Set -> SvSet,
      op empty -> empSvSet})
}
```

What follows `SERVICE`, namely `{sort Elt -> Service}`, is the *view* used here saying that `Elt` is replaced with `Service` in the instantiation of `SET` with `SERVICE`. What follows `*` is renaming. `Set` and `empty` are renamed as `SvSet` and `empSvSet`, respectively. Other operators are used without renaming. The instantiated `SET` with `SERVICE` in which `Set` and `empty` are renamed as mentioned is imported with `protecting` in `SV-SET`. In this case, `SET` is called a *template module* and `TRIV` is called a *parameter module*. Note that a template module is not always a parameterized module. Template modules with no parameters will be explained in Section 5.1.

Command `open` make a given module, `SV-SET` in this case, available.

```
open SV-SET .
 reduce #(sv(1,closed) sv(2,open)) . -- to 2.

 op svs : -> SvSet .
 reduce #(sv(1,closed) svs) = # svs + 1 . -- to true.
close
```

In `SV-SET`, `(sv(1,closed) sv(2,open))` is a term of sort `SvSet` and represents a set of services consists of two elements. Thereby, `#(sv(1,closed) sv(2,open))` is a term of `Nat` which reduces to `2` using equations of `SET` as left-to-right rewrite rules. When `svs` is a term of sort `SvSet`, `(sv(1,closed) svs)` is also a term of sort `SvSet` which represents a set of services including at least one `closed` service where `svs` represents the rest of the set. Thus, `#(sv(1,closed) svs)` reduces to `# svs + 1`.

3.2 Transition Rules

Let us consider the following module:

```
module! UPDATE {
 using(SV-SET)

 [State]
 op < _ > : SvSet -> State {constr}
 var SVS : SvSet
 var N : Nat

 trans [c2o]:
  < sv(N,closed) SVS > => < sv(N,open) SVS > .

 ctrans [o2r]:
  < sv(N,open) SVS > => < sv(N,ready) SVS >
  if # SVS > 0 .
}
```

Module `UPDATE` specifies a *state machine*. We say a "global state" as a state of the state machine in order to avoid the confusion with local states of services. A ground term of sort `State` represents a global state consisting of a set of services, where the set { < *svs* > | *svs* is a ground term of `SvSet`} represents the state space. Two transition rules, labeled by `c2o` and `o2r`, define the state transition over the global states. Transition rule `c2o` specifies that a `closed` service appearing in a global state is changed to `open`, and `o2r` specifies that an `open` service is changed to `ready` if there is at least one other service, where `ctrans` means "conditional trans".

Command `select` is similar to `open` except that it does not allow to declare new sorts, operators, equations, and so on. Command `execute` makes CafeOBJ try to apply transition rules until no one can be applied.

```
select UPDATE .
 execute < sv(1,closed) sv(2,open) > .
   -- to < sv(1,ready)  sv(2,ready) > .

 execute < sv(1,closed) > .
   -- to < sv(1,open) > .
```

Rule `c2o` makes global state `< sv(1,closed) sv(2,open) >` transit to `< sv(1,open) sv(2,open) >` then rule `o2r` makes transit it to `< sv(1,ready) sv(2,open) >` and successively makes it transit to `< sv(1,open) sv(2,open) >`. On the other hand, only rule `c2o` can be applied to global state `< sv(1,closed) >` because it has only one element.

3.2.1 Formalization of State Machines in CafeOBJ

This section summarizes the formal definitions of state machines in CafeOBJ . Please refer to [7] for detailed definitions.

Definition 1 [transition rule] Let `State` be a sort of global states, l and r be terms of sort `State`, and let c be a term of sort `Bool`, then a triple $R = [l, r, c]$ is called a transition rule and represented as "`ctrans` l `=>` r `if` c `.`" (or "`trans` l `=>` r `.`" when c is `true`).

Definition 2 [transition] Let St be a set of global states (i.e. ground terms of sort `State`), *Rule* be a set of transition rules, then a pair of global states $(S, S') \in Tr \subseteq St \times St$ is called a transition specified by *Rule* iff there exists a transition rule $R = [l, r, c] \in Rule$ and some ground substitution σ such that $S = l\sigma$, $S' = r\sigma$, and $c\sigma$ reduces to *true*. We also say R can be applied to S and say S' is a next state of S.

Definition 3 [state machine] Let *Rule* be a set of transition rules, then a state machine is a triple (St, Tr, In) where St is a set of global state, $Tr \subseteq St \times St$ is a set of transitions specified by *Rule*, and $In \subseteq St$. An element of In is called an initial state.

Definition 4 [transition sequence] Let (St, Tr, In) be a state machine, then a transition sequence is a sequence of global states $(S_0, S_1, \ldots, S_n)$ where each adjacent pair $(S_i, S_{i+1}) \in Tr$.

Notation 1 $[S\alpha, \alpha S, \alpha\beta]$ Let S be a global state and $\alpha = (S_0, S_1, \ldots, S_n)$ be a transition sequence, then $S\alpha$ is the transition sequence such that $S\alpha = (S, S_0, S_1, \ldots, S_n)$. αS is the transition sequence such that $\alpha S = (S_0, S_1, \ldots, S_n, S)$. Let $\alpha = (S_0, S_1, \ldots, S_n)$ and $\alpha = (S_{n+1}, S_{n+2}, \ldots, S_{n+m})$ be transition sequences, then $\alpha\beta$ is the transition sequence such that $\alpha\beta = (S_0, S_1, \ldots, S_n, S_{n+1}, S_{n+2}, \ldots, S_{n+m})$.

Definition 5 [reachable] Let (St, Tr, In) be a state machine, then a global state $S \in St$ is reachable iff there exists a transition sequence $(S_0, S_1, \ldots, S_n)$ where $S_0 \in In$ and $S = S_n$. Note that $S_0 \in In$ is reachable because (S_0) is a transition sequence with $n = 0$.

Definition 6 [invariant] Let (St, Tr, In) be a state machine, then a global state predicate p is an invariant iff $p(S) = true$ holds for any reachable global state S.

3.3 Search Predicates

What is called *search predicates* can be used to conduct reachability analysis for such state machines specified in CafeOBJ :

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

Let us consider the following code fragment:

```
select UPDATE .
 reduce < sv(1,closed) sv(2,open) > =(*,1)=>+ < SVS > .  -- to true.
 reduce < sv(3,closed) sv(4,ready) > =(*,1)=>+ < SVS > . -- to true.
 reduce < sv(5,open) > =(*,1)=>+ < SVS > .               -- to false.
```

By reducing the term in the code fragment, CafeOBJ finds any next states of the given global state, such as `< sv(1,open) sv(2,open) >`[3]. The first reduction returns true because both transition rules are applicable. The second one also returns true but only rule `c2o` is applicable. The third one returns false.

CafeOBJ can find next states of a given global state such that some conditions hold in those next states. Let us consider the following code fragment[4]:

[3] `*`, `1`, and `+` specify the range of search. If `2` is used instead of `*`, CafeOBJ tries to find at most two next states. If `3` is used instead of `1`, CafeOBJ finds all global states reachable from the given global state with at most three state transitions. If `*` is used instead of `+`, CafeOBJ also includes the given global state as a search target. Only `=(*,1)=>+` is used in this paper.

[4] Since the final part of the `reduce` sentence, `{ true }`, is for debugging, please ignore it.

```
open UPDATE .
 pred anyOpen : SvSet .
 eq anyOpen(sv(N,open) SVS) = true .
 var CC : Bool .
 reduce
  < sv(1,closed) sv(2,open) > =(*,1)=>+ < SVS > if CC
    suchThat CC implies anyOpen(SVS) { true } .        -- to true.
```

The reduction returns true in which CafeOBJ finds any next states of the given global state such that at least one `open` service is appearing. In this case, transition rule `c2o` makes such next state. Note that when the conditional search predicate tries a transition rule, it binds the rule's condition to Boolean variable `CC` placed at `if` clause. The `suchThat` clause uses `CC` to check `anyOpen(SVS)` only when the rule is applied.

On the other hand, when we want to check some condition holds in all possible next states, we need some trick. The following code fragment checks whether all possible next states of global state `< sv(1,closed) sv(2,open) >` include at least one `open` services:

```
reduce not (
 < sv(1,closed) sv(2,open) > =(*,1)=>+ < SVS > if CC
   suchThat not ((CC implies anyOpen(SVS)) == true) { true } ) .
                                                    -- to false.
```

This style of coding is we call the *double negation idiom* because it returns true when it CANNOT find any next states of the given global state such that NO open service is appearing. The reduction proceeds as follows:

1. Try to match LHS of `c2o` to the given global state.
2. Also try to match the rule's condition (i.e. `true` because the rule is unconditional) to `CC` and the substituted RHS (i.e. `< sv(1,open) sv(2,open) >`) to `< SVS >`.
3. Evaluate the substituted `suchThat` clause which reduces to false because `anyOpen(sv(1,open) sv(2,open))` reduces to true.
4. Then, continuing the search to find a next state where the `suchThat` clause holds, try to match LHS of `o2r` to the given global state, the condition (i.e. `# SVS > 0`) to `CC`, and the substituted RHS (i.e. `< sv(2,ready) sv(1,closed) >`) to `< SVS >`.
5. Evaluate the substituted `suchThat` clause which reduces to true because `sv(2,ready) sv(1,closed)` does not include any open services.
6. Then the search predicate returns true and the whole term reduces to false.

This means that there is a next states of global state `< sv(1,closed) sv(2,open) >` which does not include any `open` services; that is global state `< sv(1,closed) sv(2,ready) >`.

Note that this is a typical example where we need `_ == true`. In CafeOBJ , *term*1 == *term*2 reduces to `true` if both terms are reduced to be the same term and to `false` otherwise. On the other hand, *term*1 = *term*2 reduces to `true` iff *term*1 == *term*2 reduces to `true`. The following code fragment shows difference between `_ = _` and `_ == _` .

```
reduce anyOpen(sv(1,closed)) = true .
                             -- no reduction occurs.
reduce anyOpen(sv(1,closed)) == true .
                             -- reduce to false.
```

In this case, CafeOBJ cannot decide `anyOpen(SVS)` does or does not hold because the definition of `anyOpen` is incomplete and thus the first reduction above can reduce to neither `true` nor `false`. The second one using `_ == true` reduces to `false`, which is the reason why `suchThat` clause in the double negation idiom works as we intended.

3.3.1 Formalization of Search Predicates

This section describes the search predicates more formally.

Definition 7 [unconditional search predicate] Let *Rule* be a set of transition rules and let S and S' be terms of sort `State`. The unconditional search predicate, $FS(S, S')$ is represented as "S `=(*,1)=>+` S'" and holds iff there exists a transition rule $R = [l, r, c] \in Rule$ and a substitution σ such that $S\sigma = l\sigma$ holds, $S'\sigma = r\sigma$ holds, and $c\sigma$ reduces to true.

Definition 8 [conditional search predicate] Let *Rule* be a set of transition rules, S and S' be terms of sort `State`, and CC and B be terms of sort `Bool`. The conditional search predicate, $CFS(S, S', CC, B)$ is represented as "S `=(*,1)=>+` S' `if` CC `suchThat` B { $debug\_info$ }" and holds iff there exists a transition rule $R = [l, r, c] \in Rule$ and a substitution σ such that $S\sigma = l\sigma$ holds, $S'\sigma = r\sigma$ holds, $CC\sigma = c\sigma$ holds, and $B\sigma$ reduces to true. B typically has a form "CC `implies` $p(S, S')$" where $p(S, S')$ is a predicate of global states.

3.4 Verification by Proof Scores

A *proof score* is an executable specification in CafeOBJ such that if executed as expected, then the desired theorem is proved [8]. Verification by proof scores is an interactive developing process to think through meaning of the specification that is very important aspect of developing trusted systems.

For example, let us verify that in module `UPDATE` there should be a next state of global state S when at least two services included in S are not `ready`.

```
module! ProofUPDATE {
 protecting(UPDATE)

 -- Theorem to be proved.
 pred theorem : Nat LocalState Nat LocalState SvSet

 vars N N1 N2 : Nat
 vars Lst1 Lst2 : LocalState .
 var SVS : SvSet
 var SS : State

 eq theorem(N1,Lst1,N2,Lst2,SVS)
   = ((Lst1 == ready) = false and (Lst2 == ready) = false)
     implies < sv(N1,Lst1) sv(N2,Lst2) SVS > =(*,1)=>+ SS .

 -- Axiom of Nat
 eq (1 + N > 0) = true .

 -- Arbitrary constants.
```

```
 ops st1 st2 : -> LocalState
 ops n1 n2 : -> Nat
 op svs : -> SvSet
}
```

Module ProofUPDATE gets ready for verification; it defines the theorem to be proved and declares several arbitrary constants. Note that we require an axiom for natural numbers which says that the successor of a natural number is always greater than 0.

Firstly, we begin with the most general case where all aruguments of theorem are arbitrary constants:

```
-- The most general case.
open ProofUPDATE .
 reduce theorem(n1,st1,n2,st2,svs) . -- to false.
close
```

This case is too general for CafeOBJ to find any next states. We should split the case into cases which collectively cover the general case. There are three case; (1) both services are closed, (2) both services are open, and (3) one service is closed and another is open. The following is a proof score for the three cases.

```
-- Case 1: Both services are closed.
open ProofUPDATE .
 eq st1 = closed .
 eq st2 = closed .
 reduce theorem(n1,st1,n2,st2,svs) . -- to true.
close

-- Case 2: Both services are open.
open ProofUPDATE .
 eq st1 = open .
 eq st2 = open .
 reduce theorem(n1,st1,n2,st2,svs) . -- to true.
close

-- Case 3: A closed service and an open service.
open ProofUPDATE .
 eq st1 = closed .
 eq st2 = open .
 reduce theorem(n1,st1,n2,st2,svs) . -- to true.
close
```

Verification is successfully done because all cases collectively covering the most general case are proved.

3.5 Constructor-based Inductive Theorem Prover (CITP)

As described above, interactive theorem proving is a systematic process to split general cases into collectively covering cases until all cases are specific enough to be proved. Thereby, a proof score should be written more carefully when case splitting becomes deeper. It sometimes

causes to carelessly forget some cases to be proved. In fact, it may take considerable time to convince that the three cases in the previous section collectively cover all cases.

In order to assist to develop proof scores which are more systematic and easier to understand, CafeOBJ provides CITP method consisting of several special commands. The following is the list of part of CITP commands[5]:

- `:goal {eq` *term* `= true .}`
  Define the goal to be proved and let it be the current case. Multiple goal equations can be specified.

- `:ctf {eq` *LHS* `=` *RHS* `.}`
  Split the current case into two cases adding `eq` *LHS* `=` *RHS* `.` to one case and `eq (`*LHS* `=` *RHS*`) = false .` to another.

- `:csp {eq` LHS_1 `=` RHS_1 `. eq` LHS_2 `=` RHS_2 `. ...}`
  Split the current case into cases adding `eq` LHS_i `=` RHS_i `.` to each case.

- `:apply (rd)`
  Reduce the goal in the current case.

- `:def` *name* `= :ctf {...}`
  `:def` *name* `= :csp {...}`
  Name the case splitting tactic.

- `:apply (`$name_1$ $name_2$`)`
  Combine named case splitting tactics. When tactic $name_1$ splits an case into n cases and tactic $name_2$ splits into m cases, the current case is split into totally $n \times m$ cases. It can also specify tactic `rd`, i.e. `:apply (`n_1 n_2 `rd)`, which means reducing the goal in every split case.

- `:init [`*label*`] by {` *substitution* `}`
  Introduce a *labeled* lemma proven by other proof scores. *substitution* specifies how to unify the lemma to the current case. Detailed examples will be explained in Chapter 6.

- `describe proof`
  Describe the proof tree consisting of split cases. Proven cases are shown by "*" marks.

- `show proof`
  Summarize the proof tree consisting of split cases. Proven cases are shown by "*" marks.

The following is a proof score of CITP version of the example in the previous section:

```
select ProofUPDATE .
:goal {
  eq theorem(n1,st1,n2,st2,svs) = true .
}
:def csp-st1 = :csp {
 eq st1 = closed .
 eq st1 = open .
```

[5]As its name suggests, CITP has capability to automatically produce inductive goals based on constructors, however we use it only for management of proof trees in this paper.

```
 eq st1 = ready .
}
:def csp-st2 = :csp {
 eq st2 = closed .
 eq st2 = open .
 eq st2 = ready .
}
:apply (csp-st1 csp-st2 rd)
describe proof
```

Firstly, the goal to be proved should represent the most general case where all aruguments of `theorem` are arbitrary constants. Then, since class `LocalState` has only three constants (`closed, open` , and `ready`) as constructors in module `UPDATE`, there are three cases where `st1` (and also `st2`) is one of the three constants in each of cases. Thereby the combination of case splitting for `st1` and `st2` collectively covers all cases.

The final command, `describe proof`, describes the proof tree as follows:

```
==> root*
    -- context module: #Goal-root
    -- targeted sentence:
      eq theorem(n1, st1, n2, st2, svs) = true .
[csp-st1] 1*
    -- context module: #Goal-1
    -- assumption
      eq [csp-st1]: st1 = closed .
    -- targeted sentence:
      eq theorem(n1, st1, n2, st2, svs) = true .
[csp-st2] 1-1*
    -- context module: #Goal-1-1
    -- assumptions
      eq [csp-st1]: st1 = closed .
      eq [csp-st2]: st2 = closed .
    -- discharged sentence:
      eq [RD]: theorem(n1, st1, n2, st2, svs) = true .
[csp-st2] 1-2*
    -- context module: #Goal-1-2
    -- assumptions
      eq [csp-st1]: st1 = closed .
      eq [csp-st2]: st2 = open .
    -- discharged sentence:
      eq [RD]: theorem(n1, st1, n2, st2, svs) = true .
[csp-st2] 1-3*
    -- context module: #Goal-1-3
    -- assumptions
      eq [csp-st1]: st1 = closed .
      eq [csp-st2]: st2 = ready .
    -- discharged sentence:
      eq [RD]: theorem(n1, st1, n2, st2, svs) = true .
[csp-st1] 2*
    -- context module: #Goal-2
    -- assumption
```

```
      eq [csp-st1]: st1 = open .
    -- targeted sentence:
      eq theorem(n1, st1, n2, st2, svs) = true .
[csp-st2] 2-1*
    -- context module: #Goal-2-1
    -- assumptions
      eq [csp-st1]: st1 = open .
      eq [csp-st2]: st2 = closed .
    -- discharged sentence:
      eq [RD]: theorem(n1, st1, n2, st2, svs) = true .
[csp-st2] 2-2*
    -- context module: #Goal-2-2
    -- assumptions
      eq [csp-st1]: st1 = open .
      eq [csp-st2]: st2 = open .
    -- discharged sentence:
      eq [RD]: theorem(n1, st1, n2, st2, svs) = true .
[csp-st2] 2-3*
    -- context module: #Goal-2-3
    -- assumptions
      eq [csp-st1]: st1 = open .
      eq [csp-st2]: st2 = ready .
    -- discharged sentence:
      eq [RD]: theorem(n1, st1, n2, st2, svs) = true .
[csp-st1] 3*
    -- context module: #Goal-3
    -- assumption
      eq [csp-st1]: st1 = ready .
    -- targeted sentence:
      eq theorem(n1, st1, n2, st2, svs) = true .
[csp-st2] 3-1*
    -- context module: #Goal-3-1
    -- assumptions
      eq [csp-st1]: st1 = ready .
      eq [csp-st2]: st2 = closed .
    -- discharged sentence:
      eq [RD]: theorem(n1, st1, n2, st2, svs) = true .
[csp-st2] 3-2*
    -- context module: #Goal-3-2
    -- assumptions
      eq [csp-st1]: st1 = ready .
      eq [csp-st2]: st2 = open .
    -- discharged sentence:
      eq [RD]: theorem(n1, st1, n2, st2, svs) = true .
[csp-st2] 3-3*
    -- context module: #Goal-3-3
    -- assumptions
      eq [csp-st1]: st1 = ready .
      eq [csp-st2]: st2 = ready .
    -- discharged sentence:
```

```
eq [RD]: theorem(n1, st1, n2, st2, svs) = true .
```

This means that the most general case (`root`) is split into three cases (`1`, `2`, and `3`) using `csp-st1` each of which is also split into three case (for example, `1-1`, `1-2`, and `1-3`) using `csp-st2`. "*" marks show the all cases are successfully proved.

Chapter 4

Models and Representations of Cloud Orchestration

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. We say "model" which means abstractly and formally specifying the structure and behavior. A specified model is represented by a formal specification language, namely CafeOBJ in this paper.

4.1 Structure Models and Representations

CloudFormation models a structure of a cloud system simply as a set of *resources* on IaaS platform of AWS. The model is called a *template* which is represented by JSON as illustrated in Fig. 2.1. A resource has an identifier and a type and includes several *properties* which may depend on other resources.

On the other hand, TOSCA's model of a cloud system is more structured to manage any types of cloud resources, as well as inside VMs, and any types of operations such as scale-out, scale-in, shutdown, and so on. A TOSCA's model, called a *topology*, is represented by XML as illustrated in Fig 2.7. A topology consists of a set of *nodes* and a set of *relationships* between nodes. A node has several *capabilities* and *requirements*. A relationship relates a requirement of a source node to a capability of a target node.

In order to cover many different kinds of models of cloud system structures, our framework provides a generic model of a cloud system structure which consists of several *classes* of *objects*. For example, in the case of CloudFormation, a cloud system consists of two classes (resource and property) of objects whereas TOSCA models that a cloud system consists of four classes (node, relationship, capability, and requirement). For a while, we explain our framework using the simple CloudFormation template shown in Fig. 2.1 and the case of TOSCA topologies will be explained in Chapter 7.

An object has a *type*[1], an *identifier*(ID), a *local state*, and possibly *links* to other objects. In the case of the example shown in Fig. 2.1, a resource object whose type is AWS::EC2::Instance has its ID as MyInstance. The type of MyEIP resource is AWS::EC2::EIP. MyEIP has a property

[1]Do not think a *type* is that of programming languages which is called *sort* in CafeOBJ . A type is just an attribute of an object. We use the term because both CloudFormation and TOSCA use it.

but its ID is hidden and we assume it is MyEIP::InsID since its parent is MyEIP and its type is InstanceId. MyEIP::InsID has a link to MyInstance. Local states of objects are used for automation of operations, which will be explained in Section 4.2.

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 decides the set of possible types, the set of possible local states of its objects. A class also decides how its objects link to other objects.

Users of the framework should design representation of the system model in CafeOBJ language. A class is represented as a CafeOBJ module that defines a sort of its objects, a 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.

For the example show in Fig. 2.1, three objects may be represented as the following ground terms:

```
res(ec2Instance, myInstance, initial)
res(ec2Eip, myEIP, initial)
prop(instanceId,myEIP::InsID,notready,myEIP,myInstance)
```

Although the users of the framework can freely design the representation of objects, typically the constructor name represents the class of the object (`res`, `prop`), the first argument is its type (`ec2Instance`, `ec2Eip`, `instanceId`), the second is its identifier (`myInstance`, `myEIP`, `myEIP::InsID`)[2], and the third is its local state. The fourth argument of the property object represents a link to its parent, `myEIP`, and the fifth represents that the property depends on `myInstance`. The example of CafeOBJ modules representing resource and property classes will be shown in Chapter 5. Note that a link is represented by an identifier of the linked object in our framework.

4.2 Behavior Models and Representations

The framework models the behavior of an automated operation of a cloud system as a state machine 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 of some class. A transition rule makes a global state transit to another global state where local states or links of some objects are changed.

In the case of a template of CloudFormation, a global state consists of finite number of resources and their properties. CloudFormation tries to start all resources according to the dependency specified by the template. In this paper, we use a very simple behavior model of CloudFormation as an example; a local state of a resource is firstly *initial* and becomes *started* but a dependent resource can be *started* after all resources it depends become *started*. The dependency is specified such that a property linking some resource is firstly *notready* and becomes *ready* when the linked resource is *started* and a resource can be *started* when all of its properties become *ready*. Figure 4.1 illustrates the model where solid arrows show changes of local states and dashed arrows show transition rules.

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

[2]In this paper, we often use an identifier to designate an object which has the identifier for the sake of brevity.

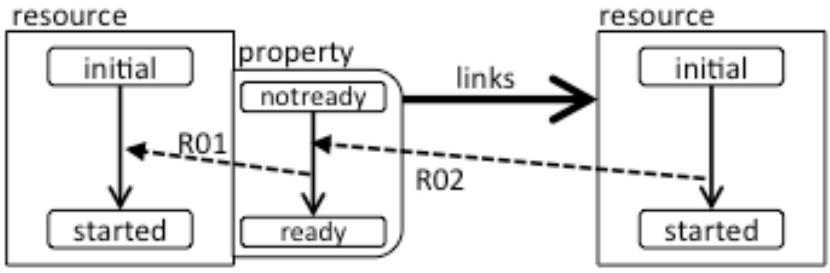

Figure 4.1: Simple Behavior Model of CloudFormation

CloudFormation, sort `State` is defined as a pair of a set of resources and a set of properties and the global state shown in Fig. 2.1 is represented as follows:[3]

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

open STATE .
 -- Constants
 ops ec2Instance ec2Eip : -> RSTypeLt .
 ops myInstance myEIP : -> RSIDLt .
 ops myEIP::InsID : -> PRIDLt .
 op instanceId : -> PRTypeLt .
 op s0 : -> State .
 eq s0 =
  < (res(ec2Instance, myInstance, initial)
     res(ec2Eip, myEIP, initial)),
    (prop(instanceId, myEIP::InsID, notready, myEIP, myInstance)) >
```

The behavior is modeled and represented by a set of two transition rules as follows:

```
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 >
```

[3]Module `LINKS` and several sorts of constants will be explained in the next chapter.

```
   => < (res(TRS,IDRS,started) SetRS), SetPR >
      if allPROfRSInStates(SetPR,IDRS,ready) .

   -- Let a not-ready property be ready
   --  if its referring resource is started.
   trans [R02]:
      < (res(TRS,IDRRS,started) SetRS),
        (prop(TPR,IDPR,notready,IDRS,IDRRS) SetPR) >
   => < (res(TRS,IDRRS,started) SetRS),
        (prop(TPR,IDPR,ready   ,IDRS,IDRRS) SetPR) > .
}
```

Predicate `allPROfRSInStates(SetPR,IDRS,ready)` checks a set of properties `SetPR` whether every property of resource `IDRS` is `ready`, which will be explained in Section 5.2. Thus, rule `R01` means that an `initial` resource becomes `started` when all of its properties are `ready`. The LHS of rule `R02` includes a resource and a property. The second link of the property is the identifier of the resource, which means the property refers the resource. Thereby, rule `R02` means that a `notready` property becomes `ready` when it refers a `started` resource.

4.3 Simulation of Models

CafeOBJ provides `execute` command to execute a state machine trying to apply transition rules as long as possible.

```
open STATERules .
 -- Constants
 ops ec2Instance ec2Eip : -> RSTypeLt .
 ops myInstance myEIP : -> RSIDLt .
 ops myEIP::InsID : -> PRIDLt .
 op instanceId : -> PRTypeLt .
 op s0 : -> State .
 eq s0 =
   < (res(ec2Instance, myInstance,initial)
      res(ec2Eip,myEIP,initial)),
     (prop(instanceId,myEIP::InsID,notready,myEIP,myInstance)) > .

 execute s0 .
 -- will be produced
 -- < (res(ec2Instance, myInstance,started)
 --    res(ec2Eip,myEIP,started)),
 --   (prop(instanceId,myEIP::InsID,ready,myEIP,myInstance)) > .
```

The followings are part of log messages of the execution above, which shows that firstly rule `R01` makes `myInstance` transit from *initial* to *ready*, then `R02` makes `myEIP::InsID` transit from *notready* to *ready*, and finally `R01` makes `myEIP` transit from `initial` to *started*.

```
...
1>[2] apply trial #1
-- rule: ctrans [R01]:
         (< (res(TRS,IDRS,initial) SetRS) , SetPR >)
```

```
       => (< (res(TRS,IDRS,started) SetRS) , SetPR >)
       if allPROfRSInStates(SetPR,IDRS,ready)
    { IDRS |-> myInstance,
      TRS |-> ec2Instance,
      SetRS |-> res(ec2Eip,myEIP,initial),
      SetPR |-> prop(instanceId,myEIP::InsID,notready,myEIP,myInstance)
    }
...
1>[19] match success #1
1<[19] (< (res(ec2Eip,myEIP,initial) res(ec2Instance,myInstance,initial)),
          (prop(instanceId,myEIP::InsID,notready,myEIP,myInstance)) >)
   --> (< (res(ec2Instance,myInstance,started) res(ec2Eip,myEIP,initial)),
          (prop(instanceId,myEIP::InsID,notready,myEIP,myInstance)) >)
1>[20] rule: trans [R02]:
          (< (res(TRS,IDRRS,started) SetRS),
             (prop(TPR,IDPR,notready,IDRS,IDRRS) SetPR) >)
       => (< (res(TRS,IDRRS,started) SetRS),
             (prop(TPR,IDPR,ready,IDRS,IDRRS) SetPR) >)
    { IDPR |-> myEIP::InsID,
      TPR |-> instanceId,
      IDRS |-> myEIP,
      SetPR |-> empPR,
      IDRRS |-> myInstance,
      TRS |-> ec2Instance,
      SetRS |-> res(ec2Eip,myEIP,initial)
    }
1<[20] (< (res(ec2Eip,myEIP,initial) res(ec2Instance,myInstance,started)),
          (prop(instanceId,myEIP::InsID,notready,myEIP,myInstance)) >)
   --> (< (res(ec2Instance,myInstance,started) res(ec2Eip,myEIP,initial)),
          (prop(instanceId,myEIP::InsID,ready,myEIP,myInstance)) >)
1>[21] apply trial #1
...
1>[42] match success #1
1<[42] (< (res(ec2Eip,myEIP,initial) res(ec2Instance,myInstance,started)),
          (prop(instanceId,myEIP::InsID,ready,myEIP,myInstance)) >)
   --> (< (res(ec2Eip,myEIP,started) res(ec2Instance,myInstance,started)),
          (prop(instanceId,myEIP::InsID,ready,myEIP,myInstance)) >)

(< (res(ec2Instance,myInstance,started) res(ec2Eip,myEIP,started)),
   (prop(instanceId,myEIP::InsID,ready,myEIP,myInstance)) >):State
```

Chapter 5

General Templates and Predicate Libraries

The framework uses the template mechanism of CafeOBJ to provide a general way to model cloud orchestration, predefined predicate libraries, and proved lemmas together with their proof scores.

5.1 Template Modules of Objects

Template module OBJECTBASE defines nine sorts and more than ten operators/predicates of objects, which generally and minimally defines what an object is in a class. The template can be instantiated and imported in a module for each class of objects, where the imported sorts and operators can be used just by renaming appropriately. For the example show in Fig. 2.1, following module RESOURCE describes specifications of the resource class for CloudFormation[1].

```
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,
```

[1] OBJECTBASE is a template with no parameter and is used to instantiate a new module and to rename predefined sorts/operators.

```
      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}
}
```

The following is the 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 the 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*;
  $\forall o, o' \in seto : (o \neq o' \rightarrow \texttt{id}(o) \neq \texttt{id}(o'))$.

- `#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 objects in *seto* are included in *setls*;
  $\forall o \in seto : \texttt{state}(o) \in setls$.

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

- `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*;
  $\exists o \in seto : \texttt{state}(o) \in setls$.

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.

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 since `OBJECTBASE` uses them to implement the predefined general operators[2].

Similarly, following module `PROPERTY` specifies the property class for the example show in Fig. 2.1.

```
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 refer : Property -> RSID
  eq type(prop(TPR,IDPR,SPR,IDRS1,IDRS2)) = TPR .
```

[2] `OBJECTBASE` declares and uses these operators and so `RESOURCE` only should define them by equations.

```
  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 refer(prop(TPR,IDPR,SPR,IDRS1,IDRS2)) = IDRS2 .

  -- Local States
  ops notready ready : -> PRStateLt {constr}
}
```

Firstly, 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.

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

5.2 Template Modules 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. Representing object structures by using links, instead of nesting structures, enables the framework to be easily applied to any kinds of model structures and to effectively provide a predefined set of operators/predicates.

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. For example, it assumes that a parameter module defines `Object` as a sort for linking objects, `LObject` as a sort for linked objects, `link` as a selector of `Object` which returns the identifier of linked object, and so on. 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 CloudFormation, the sort for linking objects is `Property`, the sort for linked objects is `Resource`, and the selectors are `parent` and `refer` 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 -> refer,
            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 the 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 the 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 `refer` 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 `refer`, 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 the 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 module `OBJLINKONE2ONE` provides predicates for one to one relationships between objects, which will be explained in Section 7.1.1.

5.3 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 as general as the templates and can be used for any instantiated predicates without individual proofs. Most of proved lemmas provided together with proof scores written in CafeOBJ .

5.3.1 Basic Lemmas

Lemma 1 (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 .
```

However, this style is somewhat inconvenient. Remember that CITP method tries to prove a fixed set of goals in many cases. If several lemmas are effective to different cases, we should use a complicated goal set such as:

```
:goal {
  eq (A1 implies B1) and (A2 implies B2) ... implies goal1 = true .
  eq (A1 implies B1) and (A2 implies B2) ... implies goal2 = true .
  ...
}
```

This style is not only complicated but also very expensive to execute. CafeOBJ internally represents a logical formula in the algebraic normal form (ANF), in which a formula represented as ANDed terms are XORed. For example, formula `(A implies B) implies goal` is represented as `A xor B xor goal xor (A and B) xor (A and goal) xor (A and B and goal)`. The ANF of a goal would become exponentially long along with the number of lemmas.

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 goal1 = true .
  eq goal2 = true .
  ...
}
```

Lemma 2 (Set Lemma) *Let* `S` *be a set of object,* `P` *be a predicate of an object,* `allObjP` *be 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*[3].

Corollary 1 *Let* `S` *be a set of object,* `P` *be a predicate of an object,* `someObjP` *be 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.*

Since a cloud system structure is modeled as a collection of several classes of objects, proof is often split into two cases where all elements in a certain set of objects do or do not satisfy a certain condition. For example, since the condition of rule `R01` is `allPROfRSInStates(SetPR, IDRS,ready)`, proof is split into two cases; all properties of resource `IDRS` are or are not `ready`.

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`.

[3]Many proved lemmas including the Set Lemma are proved using the mathematical induction about constructors. Therefor, the user should not additionally define constructors of predefined sorts.

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 in Section 5.2, allPROfRSInStates is renamed from allZOfXInStates and thus the Set 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, it 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 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') .
}
```

5.3.2 Lemmas for Link Predicates

The framework provides many proved lemmas for predefined predicates provided by OBJLINKMANY2ONE and OBJLINKONE2ONE. This section describes two of them with example usages.

Lemma 3 (Many-2-One Lemma 07) *Let* S_X *be a set of linking objects,* S_Z *be a set of linked objects,* St_X *be a set of local states of linking objects,* St_Z *be a set of local states of linked objects, and* SX *be a local state of linking object where* SX *is not included in* St_X*. Then,* allObjInStates(S_X,St) *implies* ifXInStatesThenZInStates(S_X,St_X,S_Z,St_Z).

This lemma is represented in CafeOBJ as follows[4]:

```
vars B1 B2 : Bool
pred (_when _) : Bool Bool { prec: 64 r-assoc }
eq (B1 when B2)
   = B2 implies B1 .

var S_X : SetOfLObject
var S_Z : SetOfObject
var SX : LObjState
var St_X : SetOfLObjState
```

[4] prec: 64 means the operator precedence of when is 64 (very low) and r-assoc means it is right associative.

```
var St_Z : SetOfObjState
pred m2o-lemma07 : SetOfLObject LObjState SetOfLObjState
                   SetOfObject SetOfObjState
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) .
```

In the course of verification of the transition rule set in Section 4.2, we need an invariant which says that every `started` parent resource has `ready` properties only. It is represented as follows:

```
var SetRS : SetOfResource
var SetPR : SetOfProperty
pred inv-ifRSStartedThenPRReady : State
eq inv-ifRSStartedThenPRReady(< SetRS,SetPR >)
   = ifRSInStatesThenPRInStates(SetRS,started,SetPR,ready) .
```

In order to show the invariant property of `inv-ifRSStartedThenPRReady`, we need a lemma which says that if all resources are `initial` then `inv-ifRSStartedThenPRReady` holds. The lemma could be defined as follows:

```
var SetRS : SetOfResource
var SetPR : SetOfProperty
pred lemma1 : SetOfResource SetOfProperty
eq lemma1(SetRS,SetPR) =
  allRSInStates(SetRS,initial) implies
  ifRSInStatesThenPRInStates(SetRS,started,SetPR,ready) .
```

Although this lemma may be intuitively true, a typical pitfall of developing proof scores is regarding some lemma as intuitive and skipping to prove it, which often results in leaving critical errors in specifications. However, recalling that we get `allRSInStates` by renaming `allObjInStates` and similarly `ifRSInStatesThenPRInStates` by renaming `ifXInStatesThenZInStates`, this lemma can be obtained by renaming `m2o-lemma07` as follows:

```
var SetRS : SetOfResource
var SetPR : SetOfProperty
pred m2o-lemma07-renamed : SetOfResource SetOfProperty
eq m2o-lemma07-renamed(SetRS,SetPR)
   = allRSInStates(SetRS,initial) implies
     ifRSInStatesThenPRInStates(SetRS,started,SetPR,ready)
   when not (initial \in started) .
```

Since `not (initial \in started)` is true, the `when` clause can be omitted. This is why we use `when` instead of `implies` assuming it will be omitted when renamed. Using the Implication Lemma, this lemma can be define as follows:

```
var SetRS : SetOfResource
var SetPR : SetOfProperty
eq [m2o-lemma07]:
   (allRSInStates(SetRS,initial) and
    ifRSInStatesThenPRInStates(SetRS,started,SetPR,ready))
  = allRSInStates(SetRS,initial) .
```

Lemma 4 (Many-2-One Lemma 11) *Let* S_X *be a set of linking objects,* S_Z *be a set of linked objects,* St_X *be a set of local states of linking objects,* St_Z *be a set of local states of linked objects, and* Z *and* Z' *be linked objects where* Z *and* Z' *are identical (i.e. whose identifiers, links, and types are the same) and only their local states are different*[5]*. Then, if the local state of* Z' *is included in* St_Z*,* ifXInStatesThenZInStates(S_X,St_X,(Z S_Z),St_Z) *implies* ifXInStatesThenZInStates(S_X,St_X,(Z' S_Z),St_Z).

This lemma is represented in CafeOBJ as follows:

```
vars O1 O2 : Object
pred changeObjState : Object Object
eq changeObjState(O1,O2)
   = (id(O1) = id(O2)) and
     (link(O1) = link(O2)) and
     (type(O1) = type(O2)) .

vars Z Z' : Object
var S_X : SetOfLObject
var S_Z : SetOfObject
var St_X : SetOfLObjState
var St_Z : SetOfObjState
pred m2o-lemma11 : Object Object SetOfLObject SetOfLObjState
                                 SetOfObject SetOfObjState
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') .
```

In order to show the invariant property of inv-ifRSStartedThenPRReady above, we also need another lemma which says that inv-ifRSStartedThenPRReady keeps to hold when rule R02 is applied and makes a property transit from notready to ready. The lemma could be defined as follows:

```
vars IDRS IDRRS : RSID
var IDPR : PRID
var TPR : PRType
var SetRS : SetOfResource
var SetPR : SetOfProperty
pred lemma2 : SetOfResource PRType PRID RSID RSID SetOfProperty
eq lemma2(SetRS,TPR,IDPR,IDRS,IDRRS,SetPR)
  = ifRSInStatesThenPRInStates
    (SetRS,started,(prop(TPR,IDPR,notready,IDRS,IDRRS) SetPR),ready)
  implies
    ifRSInStatesThenPRInStates
    (SetRS,started,(prop(TPR,IDPR,   ready,IDRS,IDRRS) SetPR),ready) .
```

Again this lemma may be intuitively true because its antecedent part requires that some properties should be ready and one specific property with identifier IDPR changes its local state from notready to ready. And again this lemma can also be obtained by renaming m2o-lemma11 as follows:

[5]Exactly speaking, Z and Z' are terms of CafeOBJ representing when the same object in the model is in the different local states.

```
vars IDRS IDRRS : RSID
var IDPR : PRID
var TPR : PRType
var SetRS : SetOfResource
var SetPR : SetOfProperty
pred m2o-lemma11-renamed : SetOfResource PRType PRID
                           RSID RSID SetOfProperty
eq m2o-lemma11-renamed(SetRS,TPR,IDPR,IDRS,IDRRS,SetPR) =
  = ifRSInStatesThenPRInStates
    (SetRS,started,(prop(TPR,IDPR,notready,IDRS,IDRRS) SetPR),ready)
  implies
    ifRSInStatesThenPRInStates
    (SetRS,started,(prop(TPR,IDPR,   ready,IDRS,IDRRS) SetPR),ready)
   when (state(prop(TPR,IDPR,ready,IDRS,IDRRS)) \in ready) and
        changeObjState(prop(TPR,IDPR,notready,IDRS,IDRRS),
                       prop(TPR,IDPR,   ready,IDRS,IDRRS)) .
```

The when clause reduces to true and can be omitted. Using the Implication Lemma, this lemma can be define as follows:

```
vars IDRS IDRRS : RSID
var IDPR : PRID
var TPR : PRType
var SetRS : SetOfResource
var SetPR : SetOfProperty
eq [m2o-lemma11]:
   (ifRSInStatesThenPRInStates
    (SetRS,started,(prop(TPR,IDPR,notready,IDRS,IDRRS) SetPR),ready)
   and
    ifRSInStatesThenPRInStates
    (SetRS,started,(prop(TPR,IDPR,   ready,IDRS,IDRRS) SetPR),ready))
   =
    ifRSInStatesThenPRInStates
    (SetRS,started,(prop(TPR,IDPR,notready,IDRS,IDRRS) SetPR),ready) .
```

5.3.3 Cyclic Dependency Lemma

A rule typically produces dependency of objects. For example, rule R01 in Section 4.2 makes myEIP transit from initial to started when its property myEIP::InsID is ready, which means myEIP depends on myEIP::InsID. Similarly, rule R02 makes property myEIP::InsID depend on its referring resource myInstance.

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. Such situation is called a deadlock. For example, if myInstance had a property referring myEIP, then these two resources would be mutually dependent and no transition rule could be applied.

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 that the acyclicness is an invariant, 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).

Formalization of Dependency and Acyclicness

This section will describe a formal definition of cyclic dependency and show examples using the CloudFormation example case shown in Fig. 2.1 and transition rules `R01` and `R02` in Section 4.2.

Notation 2 [$X \in C$] Let C be a class of objects in a cloud system and X be an object the system consisting of, then we denote $X \in C$ when X is of C.

Notation 3 [$st(X, S)$] Let S be a global state of a cloud system and X be an object in S, then $st(X, S)$ is the local state of X in the context of S.

Definition 9 [can make an object transit] Let $R = [l, r, c]$ be a transition rule, C be a class of objects, S be a global state, and X be an object of C. We say R can make X transit in S iff there exists a ground substitution σ such that $S = l\sigma$, $c\sigma$ reduces to true, and $st(X, l\sigma) \neq st(X, r\sigma)$. We also say R can make X transit from $st(X, l\sigma)$ to $st(X, r\sigma)$ in S. Let s and s' be local states of C, then we say R can make an object of C transit form s to s' iff there exists a global state S such that R can make an object of C transit form s to s' in S.

Definition 10 [pre-transit local states] Let R be a transition rule and C be a class of objects, then the pre-transit local states of R for 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, if $st($`myInstance`$, S)$ is *initial* then `R01` can make `myInstance` transit from *initial* to *started* in S and thus $prels($`R01`$,$ `Resource`$)$ is { `initial` }. Note that a transition rule can make objects of more than one classes transit.

Notation 4 ($S[X/s]$) Let S be a global state, C be a class of objects, X be an object of C in S, and s be a local state of C, then $S[X/s]$ is the global state such that:

- $S[X/s]$ consists of the identical objects (i.e. identifiers and types are the same) as S,
- each link of objects in $S[X/s]$ is the same as S, and
- $st(X, S[X/s]) = s$ and $\forall X' \neq X : st(X', S[X/s]) = st(X', S)$.

This notation can specify more than one objects such that $S[X_1/s_1, X_2/s_2, \dots]$. Let Σ be a set of pairs of an object and a local state, $\Sigma = \{ (X_1, s_1), (X_2, s_2), \dots \}$, then we denote $S[\Sigma]$ as $S[X_1/s_1, X_2/s_2, \dots]$.

Let S_0 be the following global state:

```
< ( res(ec2Instance, myInstance, initial)
    res(ec2Eip, myEIP, initial) ),
  ( prop(instanceId, myEIP::InsID, notready, myEIP, myInstance) ) >
```

Let us denote an object by its identifier and let Σ_0 be a set of pairs of an object and a local state such that $\Sigma_0 =$ { (`myInstance`,`started`), (`myEIP::InsID`,`ready`)}, then $S_0[\Sigma_0]$ is the following global state:

```
< ( res(ec2Instance, myInstance, started)
    res(ec2Eip, myEIP, initial) ),
  ( prop(instanceId, myEIP::InsID, ready, myEIP, myInstance) ) >
```

Definition 11 [depends on] Let S be a global state, X and X' be objects in S, and R be a transition rule where 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 there exists a set Σ of pairs of an object and a local state such that Σ includes a pair whose first element is X', R can make X transit in $S[\Sigma]$, and Σ is minimal. Here we say "minimal" which means that there exists no subset Σ' of Σ such that R can make X transit in $S[\Sigma']$. 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, rule `R01` can make `myEIP` transit from `initial` to `started` in $S_0[\Sigma_0]$, however, there is a subset of Σ_0 such that Σ_{R01} = {(`myEIP:InsID,ready`)} where rule `R01` can make `myEIP` transit also in $S_0[\Sigma_{R01}]$. Thereby, `myEIP` depends only on `myEIP:InsID` but not on `myInstance` in S_0 w.r.t. `R01`. Similarly, when Σ_{R02} = {(`myInstance,started`)}, rule `R02` can make `myEIP::InsID` transit from `notready` to `ready` in $S_0[\Sigma_{R02}]$ and thus `myEIP::InsID` depends on `myInstance` in S_0 w.r.t. `R02`.

Definition 12 [depending set] Let X be an object, R be a transition rule, and S be a global state, then the depending set of X in 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 $DS(X, S)$, i.e. $\forall X' : (dep(X, X', S) \rightarrow X' \in DS(X, S))$, and (2) if $X' \in DS(X, S)$ and X' depends on some other object X'' in S then X'' is included in $DS(X, S)$, i.e. $\forall X', X'' : (X' \in DS(X, S) \wedge dep(X', X'', S) \rightarrow X'' \in DS(X, S))$.

Definition 13 [no cyclic dependency] Let C be a class, X be an object of C, and S be a global state. We say X is in no cyclic dependency in S, denoted $noCycle(X, S)$, iff X itself is not included in $DS(X, S)$. We also say there is no cyclic dependency of C in S, denoted $noCycle_C(S)$, iff all objects of C in S are in no cyclic dependency in S.

For example, $DS(\texttt{myEIP}, S_0)$ = { `myEIP::InsID`, `myInstance` } because `myEIP` depends on `myEIP::InsID` in S_0 w.r.t. `R01` and `myEIP::InsID` depends on `myInstance` in S_0 w.r.t. `R02`. Since the depending set of `myEIP` does not include `myEIP` itself, `myEIP` is in no cyclic dependency in S_0, and there is no cyclic dependency of `Resource` in S_0.

Lemma 5 (Cyclic Dependency Lemma) *Let S be a global state, R be a transition rule, and C be a class of objects. If there is no cyclic dependency of C in 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 $prels(R, C)$, i.e.:*

$$
\begin{aligned}
&noCycle_C(S) \wedge \exists X \in C : (st(X, S) \in prels(R, C)) \rightarrow \\
&\quad \exists O \in C : (st(O, S) \in prels(R, C)) \ \wedge \\
&\qquad \forall O' \in C : (O' \in DS(O, S) \rightarrow st(O', S) \notin prels(R, C)))
\end{aligned}
$$

Proof: Let C^R be a set of objects of C in S whose local states are included in $prels(R, C)$, i.e. $C^R = \{ O \mid O \in C \ \wedge \ st(O, S) \in prels(R, C) \}$. C^R is not empty because it includes X. If every object O in C^R has at least one object $O' \in C^R \cap DS(O, S)$ then there should be some object O in C^R such that $O \in DS(O, S)$ because DS is transitive and C^R is finite. However, it means

there is cyclic dependency of C in S. □

For example, let S_0 be a global state shown above, then there is no cyclic dependency of `Resource` in S_0 and there exists `myEIP` whose local state is *initial*. Thereby, the Cyclic Dependency Lemma ensures that there exists a `Resource` object whose local state is *initial* and whose depending set includes no initial `Resource` objects, which is `myInstance`.

Focusing on One Class

When using the Cyclic Dependency Lemma, we can usually focus on one class of objects. In the CloudFormation example case, we can focus on `Resource` objects and not on `Property` objects; we should consider no cyclic dependency of only `Resource` objects and existence of a `Resource` object whose local state is in $prels$(`R01,Resource`), i.e. is `initial`. The following is a modified version of the formalization focusing on one class.

Definition 14 [depending set of the same class as] Let C be a class , X be an object of C, R be a transition rule, and S be a global state, then the depending set of the same class as X in S, denoted $DS_C(X, S)$, is defined as $DS_C(X, S) = \{ X' \in C \mid X' \in DS(X, S) \}$

Lemma 6 *Let C be a class, X be an object of C, and S be a global state. If X itself is not included in $DS_C(X, S)$, then X is in no cyclic dependency of C in S.*

Corollary 2 *Let S be a global state, R be a transition rule, and C be a class of objects. If there is no cyclic dependency of C in 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 $DS_C(O, S)$ includes no object whose local state is included in $prels(R, C)$; typically $DS_C(O, S)$ is empty.*

Definition 15 [dependency chain] Let $X_1, X_2, \ldots, X_n$ be objects and S be a global state, then the dependency chain in S, denoted $dc([X_1, X_2, \ldots, X_n], S)$, is defined as $\forall i \in \{1 \ldots n-1\} : dep(X_i, X_{i+1}, S)$.

For example, since `myEIP` depends on `myEIP::InsID` and it in turn depends on `myInstance` in S_0, there is a dependency chain in S_0, dc([`myEIP,myEIP::InsID,myInstance`], S_0).

Definition 16 [directly depending set of the same class as] Let C be a class of objects, X be an object of C, and S be a global state. The directly depending set of the same class as X in S, denoted $DDS_C(X, S)$, is defined as $\{ X' \mid \exists dc([X, X_1, \ldots, X_n, X'], S) \wedge X' \in C \wedge \forall i \in [1 \ldots n] : X_i \notin C \}$. We also say X directly depends on X' in S when $X' \in DDS_c(X, S)$.

When X and X' are objects of 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. For example, DDS_C(`myEIP`, S_0) = { `myInstance` } since there is a dependency chain dc([`myEIP,myEIP::InsID,myInstance`], S_0).

Corollary 3 *Let S be a global state, R be a transition rule, and C be a class of objects. If there is no cyclic dependency of C in 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 $DDS_C(O, S)$ includes no object whose local state is included in $prels(R, C)$; typically $DDS_C(O, S)$ is empty.*

Using a Template Module to Represent $noCycle_C$

Using the formalization of cyclic dependency explained above, the framework provides a predicate, `noCycle(S)`, which checks there is no cyclic dependency in the given global state S. Predicate `noCycle` 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 noCycle : State
  pred noCycle : Object State
  pred noCycle : SetOfObject SetOfObject State

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

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

  eq noCycle(empObj,V,S)
     = true .
  eq noCycle((O OS),V,S)
     = if O \in V then false else noCycle(DDSC(O,S),(O V),S) fi
       and noCycle(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 objects of the specific class we focus; that is the resource class in the CloudFormation example case. 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 `noCylce`. Given a global state `S`, predicate `noCycle(S)` transitively visits objects in directly depending sets `DDSC(O,S)` and checks not to find any objects already visited.

In the CloudFormation example case, getAllObjInState and DDSC can be defined as follows:

```
module! STATECyclefuns {
  protecting(STATE)

  var RS : Resource
  var SetRS : SetOfResource
  var SetPR : SetOfProperty

  op getAllRSInState : State -> SetOfResource
  eq getAllRSInState(< SetRS,SetPR >) = SetRS .

  op DDSC : Resource State -> SetOfResource
  eq DDSC(RS,< SetRS,SetPR >)
    = if state(RS) = initial then
        getRRSsOfPRsInStates(SetRS,
                             getPRsOfRSInStates(SetPR,RS,notready),
                             initial)
      else empRS fi .
}
```

Remember that rule R01 can make an initial resource transit when all of its properties are ready and that rule R02 can make a notready property when its parent is started. As explained in Section 5.2, getPRsOfRSInStates(SetPR,RS,notready) returns a set of properties which are included in the set SetPR of properties, whose parents are the resource RS, and whose local states are notready. And getRRSsOfPRsInStates(SetRS,*setPR*,initial) returns a set of resources which are included in the set SetRS of resources, which are referred by one of the properties in the set *setPR*, and whose local states are initial. Thereby, $DDS_C(X, S)$ can be defined by combining these operators.

Using getAllRSInState and DDSC as parameters template module CYCLEPRED can be instantiated as follows:

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

Resource, SetOfResource, empRS, and getAllRSInState are specified as actual parameters where DDSC is not specified because the name is the same as the formal parameter. noCycle is renamed as noRSCycle.

Lemmas for Proving Acyclicness is an Invariant

In order to use the Cyclic Dependency Lemma, the user of the framework should prove that $noCycle_C$ is an invariant, 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. For example, when rule `R01` is applied to a global state, it makes a resource object transit from `initial` to `started`. If the resource object is referred by some `notready` property object, then the property depends on the resource in the global state w.r.t. `R02` and does not depend on it in the next state. Similarly, when rule `R02` is applied, it makes a property object transit from `notready` to `ready` and the dependency between the property and its parent resource disappears. Thereby, when these rules are applied, the depending sets will become smaller than in the previous global states.

Lemma 7 (Depending Subset Lemma) *Let S and S' be global states. If $DS(X, S') \subseteq DS(X, S)$ for all objects X in S, then $noCycle(S) \rightarrow noCycle(S')$.*

Proof: $noCycle(S)$ means that $X \notin DS(X, S)$ for any X in S, which implies that $X \notin DS(X, S')$ because $DS(X, S') \subseteq DS(X, S)$. □

Corollary 4 *Let C be a class of objects and 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 5 *Let C be a class of objects and 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 $noCycle_C(S) \rightarrow noCycle_C(S')$.*

Lemma 8 (Many-2-One Lemma 24) *Let* `S_X` *be a set of linking objects,* `S_Z` *be a set of linked objects,* `St_X` *be a set of local states of linking objects, and* `X` *and* `X'` *be linking objects where* `X` *and* `X'` *are identical and only their local states are different Then, if the local state of* `X'` *is not included in* `St_X`*,* `getXsOfZsInStates((X' S_X),S_Z,St_X)` *is a subset of or equal to* `getXsOfZsInStates((X S_X),S_Z,St_X))`.

This Many-2-One Lemma 24 is represented in CafeOBJ as follows:

```
vars X X' : LObject
var S_X : SetOfLObject
var S_Z : SetOfObject
var St_X : SetOfLObjState
pred m2o-lemma24 : LObject LObject SetOfLObject
                   SetOfObject SetOfLObjState .
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 id(X) = id(X') and not state(X') \in St_X .
```

In the CloudFormation example case, in order to show the invariant property of `noRSCycle`, the corollary of the Depending Subset Lemma ensures that we should only prove that `DDSC` becomes a subset of itself when rule `R01` or `R02` is applied. It then requires another lemma which says that `getRRSsOfPRsInStates` becomes a subset of itself when rule `R01` is applied and makes a resource transit from `initial` to `started`. The lemma could be defined as follows:

```
var IDRS : RSID
var TRS : RSType
var SetRS : SetOfResource
var SetPR : SetOfProperty
pred lemma3 : RSType RSID SetOfResource SetOfProperty
```

```
eq lemma3(TRS,IDRS,SetRS,SetPR)
  = subset(getRRSsOfPRsInStates((res(TRS,IDRS,started) SetRS),
                                SetPR,initial),
           getRRSsOfPRsInStates((res(TRS,IDRS,initial) SetRS),
                                SetPR,initial) .
```

As explained in Section 5.2, `getRRSsOfPRsInStates` is renamed from `getXsOfZsInStates` and thus this lemma can be obtained by renaming `m2o-lemma24` as follows:

```
vars RS RS' : Resource
var SetRS : SetOfResource
var SetSRS : SetOfRSState
var SetPR : SetOfProperty
ceq [m2o-lemma24]:
  subset(getRRSsOfPRsInStates((RS' SetRS),SetPR,SetSRS),
         getRRSsOfPRsInStates((RS  SetRS),SetPR,SetSRS))
  = true
  if id(RS) = id(RS') and not state(RS') \in SetSRS .
```

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* defined above.

Notation 5 [$rel(X, X', S)$] Let S be a global state and X and X' be objects in S. When there is some relationship r between X and X', we denote it as $\underline{r(X, X', S)}$. Note that "X depends on X' in S" is one of such relationships.

Definition 17 [directly relating set] Let S be a global state, X be an object in S, and r be a relationship of objects. Then, the directly relating set of X in S w.r.t. r, denoted $\underline{DRS_r(X, S)}$, is defined as $DRS_r(X, S) = \{ X' \mid \overline{r(X, X', S)} \}$.

Definition 18 [relating set] Let X be an object, S be a global state, and r be a relationship of objects, then the relating set of X in S w.r.t. r, denoted $\underline{RS_r(X, S)}$, is recursively defined as (1) $\forall X' : (\overline{r(X, X', S) \rightarrow X' \in RS_r(X, S)})$, and (2) $\forall X', X'' : \overline{(X' \in RS_r(X, S) \wedge r(X', X'', S) \rightarrow X'' \in RS_r(X, S))}$.

Definition 19 [no cyclic relationship] Let X be an object, S be a global state, and r be a relationship of objects, then we say X is in no cyclic relationship in S w.r.t. r, denoted $\underline{noCycle_r(X, S)}$, iff X itself is not included in $\overline{RS_r(X, S)}$.

Lemma 9 *Let C is a class, X be an object of C, S be a global state, and r be 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, i.e.:*

$$\forall X : DDS_c(X, S) \subseteq DRS_r(X, S) \rightarrow \forall X : (noCycle_r(X, S) \rightarrow noCycle_C(X, S))$$

Proof: $DDS_c(X, S) \subseteq DRS_r(X, S)$ means that $DS_c(X, S) \subseteq RS_r(X, S)$. Thereby, if $noCycle_c(X, S)$ does not holds, then $X \in DS_C(X, S)$ and $X \in RS_r(X, S)$, which is a contradiction. □

This lemma allows the user of the framework to define DDSC implementing some simpler relationship r instead of the true DDS_C and use noCycle defined by using the DDSC instead of the true $noCycle_c$. For example, when we adopt the constraint of no cyclic chains of links in the CloudFormation example case, DDSC can be simply defined as follows:

```
var RS : Resource
var SetRS : SetOfResource
var SetPR : SetOfProperty
eq DDSC(RS,< SetRS,SetPR >)
  = getRRSsOfPRs(SetRS,getPRsOfRS(SetPR,RS)) .
```

However noCycle defined by using the simpler DDSC above is not the true $noCycle_C$, we can use the Cyclic Dependency Lemma if we can prove the invariant property of noCycle.

Chapter 6

Verification Procedure of Leads-to Properties

The framework provides an overall verification procedure for a kind of liveness properties, *leads-to* properties adopted from UNITY logic [3], as well as invariant properties. The procedure assists the users of the framework to systematically think and develop proof scores for verification of cloud orchestration.

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. We say "surely" to mean that the system always reaches some final state from any initial states. This kind of reachability is one of the most important properties of practical automation of cloud systems.

Futatsugi [7] defines leads-to property based on transition sequences of state machines and proposes a set of sufficient conditions for it as follows:

Definition 20 [p `leads-to` q] Let $TS = (St, Tr, In)$ be a state machine, p and q be predicates of St, $St^R \subseteq St$ be the set of reachable states of TS, and Θ be the set of transition sequences of TS, then $\underline{p \texttt{ leads-to } q}$ defined as follows:

$$\begin{aligned}\forall S\,\alpha \in \Theta : (S \in St^R \wedge p(S) \wedge \forall S' \in S\,\alpha : \neg q(S')\\ \rightarrow \exists T \in St, \exists \beta \in \Theta : (q(T) \wedge S\,\alpha\beta T \in \Theta))\end{aligned}$$

Lemma 10 *Let p_0, p, and q be predicates of St:* $(p_0 \rightarrow p) \wedge (p \texttt{ leads-to } q) \rightarrow p_0 \texttt{ leads-to } q$.

Lemma 11 (sufficient conditions for leads-to) *Let $TS = (St, Tr, In)$ be a state machine, p and q be predicates of St, inv be an invariant of TS, and m be a natural number function of St, then the following four conditions are sufficient for (p* `leads-to` *q) to hold.*

$$\begin{aligned}\forall (S, S') \in Tr : &\quad ((inv(S) \wedge p(S) \wedge \neg q(S)) \rightarrow (p(S') \vee q(S')))\\ \forall (S, S') \in Tr : &\quad ((inv(S) \wedge p(S) \wedge \neg q(S)) \rightarrow (m(S) > m(S')))\\ \forall S \in St : &\quad ((inv(S) \wedge p(S) \wedge \neg q(S)) \rightarrow \exists S' \in St : (S, S') \in Tr)\\ \forall S \in St : &\quad ((inv(S) \wedge p(S) \wedge (m(S) = 0)) \rightarrow q(S))\end{aligned}$$

Definition 20 of (p `leads-to` q) includes the case where there is an infinite transition sequence $\alpha = (S_0, S_1, \dots)$ such that predicate q never holds, $\forall S_i \in \alpha : \neg q(S_i)$. However, `leads-to` may be defined in a narrower sense where no such infinite transition sequence is allowed. Here, we call

the former as *weak-leads-to* and the latter as *strong-leads-to*. The set of conditions proposed by Lemma 11 is sufficient to both meanings of leads-to because the properly decreasing natural number function, m, ensures that transition sequences never become infinite while keeping q not to hold. In the rest of this paper, we mean (p `leads-to` q) as (p `strong-leads-to` q).

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 $Fn \subseteq St$ 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 $S \in Fn$.

The lemma 10 ensures that what we should do is to find a state predicate p such that ($init \rightarrow p$) and p satisfies the sufficient conditions for (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 21 [continuous predicate] The continuous predicate, *cont*, is the predicate which holds iff there exists some next state of a given state. Let $TS = (St, Tr, In)$ be a state machine, then $\forall S \in St : cont(S)$ `iff` $\exists S' \in St : (S, S') \in Tr$.

Lemma 12 (sufficient conditions for $init$ `leads-to` $final$**)** *Let $TS = (St, Tr, In)$ be a state machine, inv be a conjunction of some state predicates and m be a natural number function of St, then the following six conditions are sufficient for (init* `leads-to` $final$*) to hold.*

$$\forall S \in St : \quad (init(S) \rightarrow cont(S)) \tag{1}$$

$$\forall (S, S') \in Tr : \quad ((inv(S) \wedge \neg final(S)) \rightarrow (cont(S') \vee final(S'))) \tag{2}$$

$$\forall (S, S') \in Tr : \quad ((inv(S) \wedge \neg final(S)) \rightarrow (m(S) > m(S'))) \tag{3}$$

$$\forall S \in St : \quad ((inv(S) \wedge cont(S) \wedge (m(S) = 0)) \rightarrow final(S)) \tag{4}$$

$$\forall S \in St : \quad (init(S) \rightarrow inv(S)) \tag{5}$$

$$\forall (S, S')\ Tr : \quad (inv(S) \rightarrow inv(S')) \tag{6}$$

Proof: Let p in the "sufficient conditions for leads-to" lemma be *cont*, then $\forall (S, S') \in Tr : p(S) = true$ holds and $\forall S \in St : (p(s) \rightarrow \exists S' \in St : (S, S') \in Tr)$. □

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' such that $((cont(S') \vee final(S')) \wedge (m(S') = 0))$. Condition (4) then ensures $final(S')$ holds. Here, m is called a *state measuring function*. When condition (5) and (6) hold, each state predicate included in *inv* is called an invariant.

The rest of this chapter explains the verification procedure for six sufficient conditions above using the CloudFormation example case and the case of TOSCA topologies will be explained in Chapter 7.

6.1 Procedure: Definition of Predicates

Step 0-1: Define $init$ and $final$.
In the CloudFormation example case, predicates $init(S)$ and $final(S)$ can be represented by

CafeOBJ as follows:

```
var SetRS : SetOfResource
var SetPR : SetOfProperty
var S : State

pred init : State
eq init(< SetRS,SetPR >)
   = wfs(< SetRS,SetPR >) and
     noRSCycle(< SetRS,SetPR >) and
     allRSInStates(SetRS,initial) and
     allPRInStates(SetPR,notready) .

pred final : State
eq final(< SetRS,SetPR >)
   = allRSInStates(SetRS,started) .

pred wfs : State
eq wfs(S)
   = wfs-atLeastOneRS(S) and
     wfs-uniqRS(S)       and wfs-uniqPR(S) and
     wfs-allPRHaveRS(S)  and wfs-allPRHaveRRS(S) .

pred wfs-atLeastOneRS : State
eq wfs-atLeastOneRS(< SetRS,SetPR >) = not (SetRS = empRS) .

pred wfs-uniqRS : State
eq wfs-uniqRS(< SetRS,SetPR >) = uniqRS(SetRS) .

pred wfs-uniqPR : State
eq wfs-uniqPR(< SetRS,SetPR >) = uniqPR(SetPR) .

pred wfs-allPRHaveRS : State
eq wfs-allPRHaveRS(< SetRS,SetPR >) = allPRHaveRS(SetPR,SetRS) .

pred wfs-allPRHaveRRS : State
eq wfs-allPRHaveRRS(< SetRS,SetPR >) = allPRHaveRRS(SetPR,SetRS) .
```

Among conditions composing *init*(S), one without referring any local states of objects is called a *wfs (well-formed state)* and we usually gather them and define predicate `wfs` as a conjunction of them. The reason why we need several wfs predicates is because representing a global state as a tuple of sets of objects is too general to represent structural constraints, such as identifiers should be unique, there is no dangling link, and so on. Each structural constraint is typically represented as a wfs and should be an invariant. In addition, do not forget to include *noCycle* in the *init* predicate when using the Cyclic Dependency Lemma.

Step 0-2: Define *cont*.
Since *cont*(S) means that state S has at least one next state, it can be specified as follows using the unconditional search predicate of CafeOBJ :

```
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 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. For the CloudFormation example case, m can be defined as follows:

```
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 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.

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 (6) above and are introduced by the users of the framework. For example, the CloudFormation example case requires an invariant `inv-ifRSStartedThenPRReady` as explained in Section 5.3.

Predicate *inv* is a conjunction of all the invariants, however, the straightforward representation is not so efficient. CafeOBJ needs to internally maintain long ANDed terms and to spend much processing time. Fortunately, there is more efficient representation. Since the sufficient conditions (2), (3), (4), and (6) 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

-- wfs-*:
ceq inv(S) = false if not wfs-atLeastOneRS(S) .
ceq inv(S) = false if not wfs-allPRHaveRS(S) .
ceq inv(S) = false if not wfs-allPRHaveRRS(S) .

-- inv-*:
ceq inv(S) = false if not inv-ifRSStartedThenPRReady(S) .
```

Note that only three of six wfs predicates are used to define *inv*, since they directly take some roles in proofs.

As to sufficient conditions (5) and (6), *inv* is also included in their consequent parts, which case will be explained in Section 6.6.

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.

In the CloudFormation example case, we will introduce an `initial` resource and claim that every resource in DDS_c of the resource is not `initial`. 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 in the next section.

Step 0-6: Prepare arbitrary constants.
Proof scores for the sufficient conditions requires many arbitrary constants. In order to make the proof score be easy to understand, those constants are consistently named and defined. The following shows the definitions of constants for the CloudFormation example case:

```
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
```

6.2 Procedure: Proof of 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 arbitrary constants every of which represents an arbitrary set of objects of each class. For the CloudFormation example case, the most general case is as follows where `sRS` and `sPR` are arbitrary constants for a set of resources and properties respectively:

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

This case is too general to judge whether the condition does or does not hold. Thereby, no reduction occurs.

Step 1-2: Think which rule is applied to the global state in the current case. The rule is referred to as the *current rule*.
One of the main benefits of interactive proof development is that thinking through meaning of the specification leads to deep understanding of it. If the developer of proofs cannot find the first applied rule, it means insufficient understanding of the specification. For the CloudFormation example case, the first rule is `R01`.

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.
Since LHS of rule `R01` requires the global state to have at least one `initial` resource, the current case is split into three more cases, i.e. no resource, at least one `initial` or `started` resource. In the following proof score, `trs`, `idRS`, and `sRS'` are arbitrary constants for a type of the resource, an identifier of the resource, and a set of resources respectively.

```
:csp {
  eq sRS = empRS .
  eq sRS = (res(trs,idRS,srs) sRS') .
}
-- Case 1: When there is no resource:
:apply (rd) -- 1
-- Case 2: When there is a resource:
-- The state of the resource is initial or started.
:csp {
  eq srs = initial .
  eq srs = started .
}
-- Case 2-1: When the resource is initial:
```

```
... -- More case splitting needed.
-- Case 2-2: When the resource idRS is started:
:apply (rd) -- 2-2
```

Note that `res(trs,idRS,srs)` represents an arbitrary resource. The goal of Case 1 is proved because `wfs-atLeastOneRS(S)` does not hold and thus `init(S)` does not hold. The goal of Case 2-2 is also proved because `allRSInStates(SetRS,initial)` does not hold. Only Case 2-1 remains unproved and it then becomes the current case.

Step 1-4: Split the current case into cases where the condition of the rule does or does not hold. Since the condition of rule `R01` requires all properties of the `initial` resource are `ready`, Case 2-1 is split into two more cases; all properties are or are not `ready`. As explained in Section 5.3, the Set Lemma ensures that these cases are represented as follows where `tpr`, `idPR`, `idRRS`, and `sPR'` are arbitrary constants for a type of the property, an identifier of the property, an identifier of a resource referred by the property, and a set of properties respectively:

```
-- Case 2-1: When the resource is initial:
-- The condition of R01 is allPROfRSInStates(sPR,idRS,ready) .
:csp {
  eq allPROfRSInStates(sPR,idRS,ready) = true .
  eq sPR = (prop(tpr,idPR,notready,idRS,idRRS) sPR') .
}
-- Case 2-1-1: When all of properties of the resource are ready.
:apply (rd) -- 2-1-1
-- Case 2-1-2: When there is a not-ready property of the resource:
... -- More case splitting needed.
```

Note that `prop(tpr,idPR,notready,idRS,idRRS)` represents an arbitrary `notready` property whose parent is `idRS`. In Case 2-1-1, rule `R01` can be applied, which means `cont(S)` holds. Only Case 2-1-2 remains unproved.

Step 1-5: When there is a dangling link, split the current case into cases where the linked object does or does not exist.
In Case 2-1-2, property `idPR` has a link to a resource with identifier `idRRS`. Thereby, it is split into three more cases; a resource with identifier `idRRS` does not exist, does exist and it is `initial` or `started`. The nonexistence can be represented as predefined predicate `existObj` (renamed to `existRS` in this case) does not hold. Case 2-1-2 is split into the following three cases.

```
-- Case 2-1-2: When there is a not-ready property of the resource:
-- The resource referred by the property does or does not exist.
:csp {
  eq existRS(sRS',idRRS) = false .
  eq sRS' = (res(trs',idRRS,srs') sRS'') .
}
-- Case 2-1-2-1: When the referred resource does not exist:
:apply (rd) -- 2-1-2-1
-- Case 2-1-2-2: When the referred resource exists:
-- The state of the resource is initial or started.
:csp {
```

```
  eq srs' = initial .
  eq srs' = started .
}
-- Case 2-1-2-2-1: When the resource idRRS is initial:
... -- More consideration needed.
-- Case 2-1-2-2-2: When the resource idRRS is started:
:apply (rd) -- 2-1-2-2-2
```

The goal of Case 2-1-2-1 is proved because `wfs-allPRHaveRRS(S)` does not holds and the goal of Case 2-1-2-2-2 is proved because `allRSInStates(SetRS,initial)` does not holds. Only Case 2-1-2-2-1 remains unproved.

Step 1-6: When falling in a cyclic situation, use the Cyclic Dependency Lemma.
Since `noRSCycle` is included in the `init` condition and the resource `idRS` is `initial`, the Cyclic Dependency Lemma ensures there exists some `initial` resource `RS` such that no resource in `DDSC(RS,S)` is `initial`. Recalling that we chose `idRS` as an arbitrary `initial` resource in Step 1-3, we can assume that itself is such `RS` and can claim that there is a contradiction when its DDS_C includes any `initial` resource using a `:init` command as follows:

```
-- Case 2-1-2-2-1: When the resource idRRS is initial:
-- The Cyclic Dependency Lemma rejects this case.
:init [Cycle] by {
  T:RSType <- trs;
  IDRS:RSID <- idRS;
  S:State <- < sRS, sPR >;
}
:apply (rd) -- 2-1-2-2-1
```

The `:init` command substitutes variable `T` and `IDRS` with the type and identifier of the resource and `S` with the global state. It has the same effect as adding the following equation into the current case:

```
ceq true = false
   if someRSInStates(DDSC(res(trs, idRS, initial),< sRS, sPR >),
                     initial) .
```

Since DDS_C of the resource includes resource `res(trs',idRRS,initial)`, there is a contradiction and the goal of this case is proved.

The following is the result of a "`show proof`" command, which shows that goals of all split cases are proved and thus condition (1) is proved.

```
root*
[csp] 1*
[csp] 2*
[csp] 2-1*
[csp] 2-1-1*
[csp] 2-1-2*
[csp] 2-1-2-1*
[csp] 2-1-2-2*
[csp] 2-1-2-2-1*
[csp] 2-1-2-2-2*
[csp] 2-2*
```

Figure 6.1 summarizes the procedure.

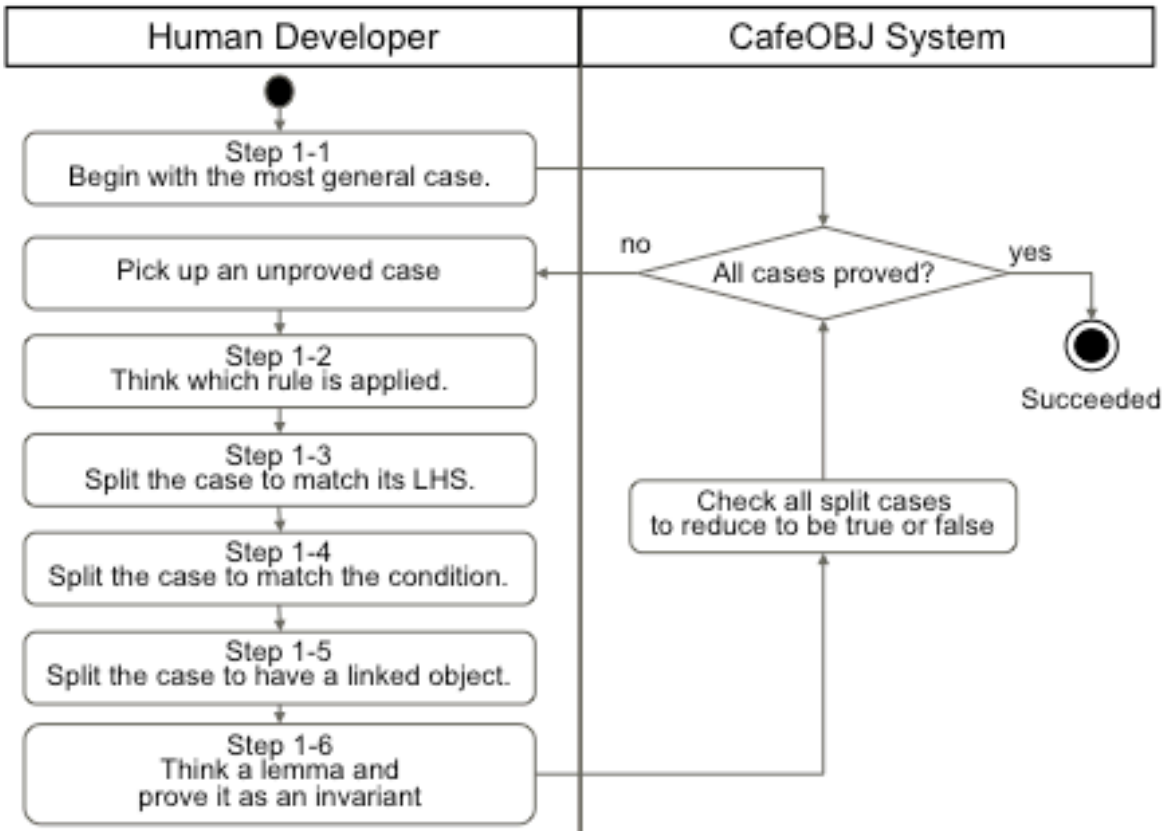

Figure 6.1: Verification Procedure for Condition (1)

6.3 Procedure: Proof of Condition (2)

Step 2-0: Define a predicate to be proved.
Using the double negation idiom in Section 3.3, 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. For the CloudFormation example case, we can begin with two cases for two rules as follows, which are too general:

```
-- Goal of Condition (2) for rule R01
```

```
:goal {
  eq contcont(< (res(trs,idRS,initial) sRS), sPR >) = true .
}

-- Goal of Condition (2) for rule R02
:goal {
  eq contcont(< (res(trs,idRRS,started) sRS),
                (prop(tpr,idPR,notready,idRS,idRRS) sPR) >) = true .
}
```

The rest of this section describes the procedure for condition (2) using the case of rule R01 as an example. The case of rule R02 will be explained in Section 6.7.

Step 2-2: Split the current case for a rule into cases where the condition of the rule does or does not hold.
Since the condition of rule R01 requires all properties of the initial resource are ready, the root case is split into two cases; all properties are or are not ready.

```
-- The condition of R01 does or does not hold for the resource of idRS.
:ctf {
  eq allPROfRSInStates(sPR,idRS,ready) = true .
}
-- Case 1: When the condition of R01 holds:
... -- More case splitting needed
-- Case 2: When the condition of R01 does not hold:
:apply (rd) -- 2
```

Remember that in Step 1-4 explained above, we used a :csp command for case splitting based on the condition of rule R01 because we need more consideration for the negative case. In Step 2-2, we can simply use a :ctf command, since Case 2 has no next state and its goal can be proved. Thereby, only Case 1 remains unproved.

Step 2-3: Split the current case into cases where predicate *final* does or does not hold in the next state.
In Case 1, rule R01 makes an initial resource transit to started and the next state becomes a final state if all other resources included the set sRS of resources are already started. Otherwise there is at least one other initial resource. Using the Set Lemma, we can split the case as follows where trs', idRS', and sRS' are arbitrary constants for a type of the resource, an identifier of the resource, and a set of resources respectively:

```
-- Case 1: When the condition of R01 holds for the resource of idRS:
-- All of the other resources are or are not started.
:csp {
  eq allRSInStates(sRS,started) = true .
  eq sRS = (res(trs',idRS',initial) sRS') .
}
-- Case 1-1: When all of the other resources are started:
:apply (rd) -- 1-1
-- Case 1-2: When there is an initial resource:
... -- More case splitting needed
```

The goal of Case 1-1 is proved because the next state is final. Case 1-2 remains unproved.

Step 2-4: Similarly as Step 1-2, think which rule can be applied to the next state. The rule is referred to as the *current rule*.
Since the next state in Case 1-2 includes an `initial` resource with identifier `idRS'`, rule `R01` can be applied to it.

Step 2-5: Similarly as Step 1-3, 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.
In this example, the next state of the current case already matches to LHS of rule `R01`.

Step 2-6: Similarly as Step 1-4, split the current case into cases where the condition of the current rule does or does not hold in the next state.
Again the Set lemma can be used similarly as Step 1-4 as follows:

```
-- Case 1-2: When there is an initial resource:
:csp {
  eq allPROfRSInStates(sPR,idRS',ready) = true .
  eq sPR = (prop(tpr,idPR,notready,idRS',idRRS) sPR') .
}
-- Case 1-2-1: When all of properties of the resource idRS' are ready.
:apply (rd) -- 1-2-1
-- Case 1-2-2: When at least one of properties is not-ready.
-- Because sPR is redefined,
-- allPROfRSInStates(sPR,idRS,ready) should be claimed again.
:set(normalize-init,on)
:init ( ceq B1:Bool = true if not B2:Bool . ) by {
  B1:Bool <- allPROfRSInStates(sPR,idRS,ready) ;
  B2:Bool <- allPROfRSInStates(sPR,idRS,ready) == true ;
}
:set(normalize-init,off)
... -- More consideration needed.
```

The goal of Case 1-2-1 is proved. Case 1-2-2 remains unproved and this is somewhat troublesome for CafeOBJ system.

Remember that in Step 2-2 we already introduced an equation which claims that every property of the resource `idRS` in the set of properties `sPR` is `ready`. Here in Case 1-2-2, we need to define that `sPR` has a `notready` property `idPR` (consequently its parent should not be the resource `idRS`) and the rest of properties are included in the set `sPR'`. This breaks the confluence property of equations; when reducing the term `allPROfRSInStates(sPR,idRS,ready)`, it reduces to true if CafeOBJ firstly uses the equation introduced in Step 2-2. But if CafeOBJ firstly uses the equation introduced here, it reduces to `allPROfRSInStates(sPR',idRS,ready)` and what we hope is the former. However we should not break the confluence property, it is a trade-off between the ideal and the consistent case splitting manner. What is more important is to keep proof scores independent from the reduction strategy of CafeOBJ system. To do so, we have to write the proof score such as it does nothing when `allPROfRSInStates(sPR,idRS,ready)` reduces to true but otherwise it claims that the term reduces to true, which is the work of the `:init` command above.

The command `:set(normalize-init,on)` means that substituted variables should be reduced to normal forms when the equation is introduced by the `:init` command; its default

option is off. When the variable B1 reduces to true, B2 also reduces to true and the equation to be introduced becomes "ceq true = true if not true ." which has no meaning because the condition part never holds. When B1 reduces to allPROfRSInStates(sPR',idRS,ready), B2 reduces to false and thus the equation to be introduced becomes as follows, which we want to claim:

```
ceq allPROfRSInStates(sPR',idRS,ready) = true if not false .
```

Step 2-7: Similarly as Step 1-5, when there is a dangling link, split the current case into cases where the linked object does or does not exist.
In Case 1-2-2, a property has a link to a resource with identifier idRRS. Thereby, it is split into three more cases; a resource with identifier idRRS does not exist, does exist and it is initial or started. The nonexistence can be represented as predefined predicate existRS does not hold. Case 1-2-2 is split into the following three cases:

```
-- Case 1-2-2: When at least one of properties is not-ready.
... -- Consideration above needed.
-- The resource referred by the property does or does not exist.
:csp {
  eq existRS(sRS',idRRS) = false .
  eq sRS' = (res(trs'',idRRS,srs'') sRS'') .
}
-- Case 1-2-2-1: When the referred resource does not exist:
:apply (rd) -- 1-2-2-1
-- Case 1-2-2-2: When the referred resource exists:
-- The state of the resource is initial or started.
:csp {
  eq srs'' = initial .
  eq srs'' = started .
}
-- Case 1-2-2-2-1: When the resource idRRS is initial:
... -- More consideration needed.
-- Case 1-2-2-2-2: When the resource idRRS is started:
:apply (rd) -- 1-2-2-2-2
```

The goal of Case 1-2-2-1 is proved because wfs-allPRHaveRRS(S) does not holds and then inv(S) does not hold as described in Section 6.1. The goal of Case 1-2-2-2-2 is also proved because the notready property idPR refers the started resource and so rule R02 is applicable in the next state. Only Case 1-2-2-2-1 remains unproved.

Step 2-8: Similarly as Step 1-6, when falling in a cyclic situation, use the Cyclic Dependency Lemma.
If the invariant property of noRSCycle is proved, we can use the Cyclic Dependency Lemma in any reachable state. In Case 1-2-2-2-1, there is an initial resource idRS' and so the lemma ensures there exists some initial resource RS such that no resource in DDSC(RS,S) is initial. Recalling that we chose idRS' as an arbitrary initial resource in Step 2-3, we can assume that itself is such RS and can claim that there is a contradiction when its DDS_C includes any initial resource using a :init command as follows:

```
-- The Cyclic Dependency Lemma rejects this case.
:init [Cycle] by {
  T:RSType <- trs';
```

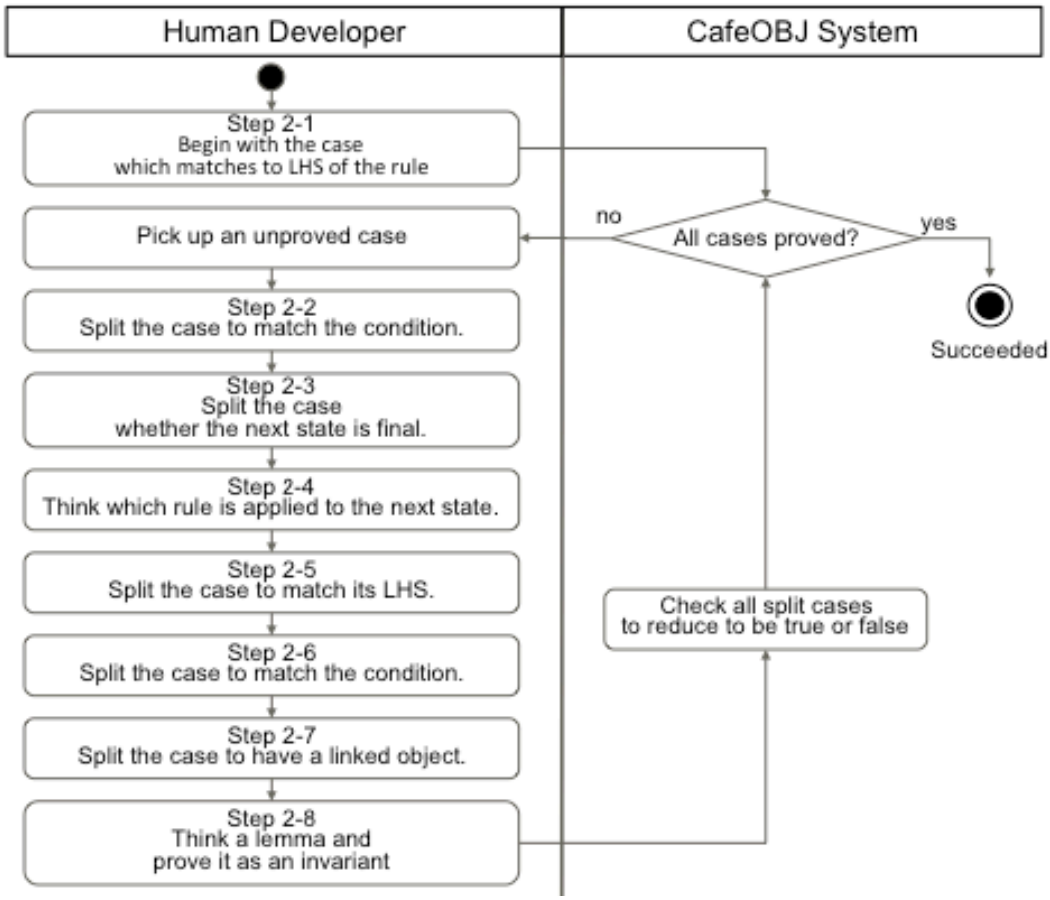

Figure 6.2: Verification Procedure for Condition (2) for each rule

```
  IDRS:RSID <- idRS';
  S:State <- < (res(trs,idRS,initial) sRS), sPR >;
}
:apply (rd) -- 1-2-2-2-1
```

The goal of this case is proved by the contradiction.

The following is the result of a "`show proof`" command, which shows that goals of all split cases are proved and thus condition (2) for rule `R01` is proved:

```
root*
[ctf] 1*
[csp] 1-1*
[csp] 1-2*
[csp] 1-2-1*
[csp] 1-2-2*
[csp] 1-2-2-1*
[csp] 1-2-2-2*
[csp] 1-2-2-2-1*
[csp] 1-2-2-2-2*
[ctf] 2*
```

Figure 6.2 summarizes the procedure for each transition rule.

6.4 Procedure: Proof of 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 knowledge to deduce natural number expressions, the framework provides several axioms to be used for proof of condition(3) and (4). The following is one of those axioms to be used for the CloudFormation example:

```
var N : Nat
eq (1 + N) > N = true .
```

Step 3-1: Define a predicate to be proved.
Using the double negation idiom in Section 3.3, 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.
Step 3-3: Split the current case for a rule into cases where the condition of the rule does or does not hold.

For the CloudFormation example case, we can begin with two cases for two rules. Since rule `R01` is conditional, the general case should be split into two cases according to Step 3-3. Then, the goals of totally three cases can be proved and thus condition (3) is proved as follows:

```
-- Goal of Condition (3) for rule R01
:goal {
  eq mesmes(< (res(trs,idRS,initial) sRS), sPR >) = true .
}
-- The condition of R01 does or does not hold for S.
:ctf {
  eq allPROfRSInStates(sPR,idRS,ready) = true .
}
-- Case 1: When the condition of R01 holds:
:apply (rd) -- 1
-- Case 1: When the condition of R01 does not hold:
:apply (rd) -- 2

-- Goal of Condition (3) for rule R02
:goal {
```

```
  eq mesmes(< (res(trs,idRRS,started) sRS),
              (prop(tpr,idPR,notready,idRS,idRRS) sPR) >) = true .
}
:apply (rd) -- goal
```

6.5 Procedure: Proof of Condition (4)

Step 4-0: Use natural number axioms.
Since the measuring function m is defined as the sum of natural numbers, $m(S) = 0$ means each of the numbers is also zero. Thereby, when either `#ResourceInStates(initial,sRS)` or `#PropertyInStates(notready,sPR)` is not zero, $m(S) = 0$ does not hold and the goal is proved. The framework provides several natural number axioms to enable such deduction as follows:

```
vars N1 N2 : Nat
var Nz : NzNat
eq (N1 + N2 = 0) = (N1 = 0) and (N2 = 0) .
eq (Nz = 0) = false .
```

Note that `NzNat` is a subsort of `Nat` which does not include `0`.

Step 4-1: Define a predicate to be proved.
Predicate `mesfinal` for condition (4) can be defined as follows:

```
var S : State
pred mesfinal : State .
eq mesfinal(S)
   = inv(S) and cont(S) and m(S) = 0 implies final(S) .
```

Step 4-2: Begin with the cases each of which matches to LHS of each rule.
Step 4-3: Split the current case for a rule into cases where the condition of the rule does or does not hold.

For the CloudFormation example case, we can begin with two cases for two rules. Since rule `R01` is conditional, the general case should be split into two cases according to Step 4-3. Then, the goals of totally three cases can be proved and thus condition (4) is proved as follows:

```
-- Goal of Condition (4)' for rule R01
:goal {
  eq mesfinal(< (res(trs,idRS,initial) sRS), sPR >) = true .
}
-- The condition of R01 does or does not hold for S.
:ctf {
  eq allPROfRSInStates(sPR,idRS,ready) = true .
}
-- Case 1: When the condition of R01 holds:
:apply (rd) -- 1
-- Case 2: When the condition of R01 does not hold:
:apply (rd) -- 2

-- Goal of Condition (4)' for rule R02
:goal {
  eq mesfinal(< (res(trs,idRRS,started) sRS),
```

```
                (prop(tpr,idPR,notready,idRS,idRRS) sPR) >) = true .
}
:apply (rd) -- goal
```

6.6 Procedure: Proof of Condition (5) & (6)

Since predicate *inv* is a conjunction of typically 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:

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

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

The rest of this section describes the proof procedure for three typical kinds of invariants in the CloudFormation example case.

Proof of Invariants for Local State Constraints

The most typical kind of invariants other that well-formed state predicates is for constraints about local states of objects. For example, `inv-ifRSStartedThenPRReady` says that every `started` parent resource has `ready` properties only. It is defined by using a predefined predicate `ifXInStatesThenZInStates` (renamed as `ifRSInStatesThenPRInStates`). Since the framework provides many lemmas for predefined predicates, it is easy to prove the invariant property of such a predicate.

Step 5-0: Define a predicate to be proved.
Predicate `initinv` for condition (5) 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-ifRSStartedThenPRReady(S) .
```

Step 5-1: Instantiate proved lemmas for predefined predicates.
As described in Section 5.3.2, proved lemma `m2o-lemma07` can be instantiated and used for proof of condition (5) as follows:

```
var SetRS : SetOfResource
var SetPR : SetOfProperty
-- Instantiating m2o-lemma07:
-- 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-lemma07]:
    (allRSInStates(SetRS,initial) and
```

```
      ifRSInStatesThenPRInStates(SetRS,started,SetPR,ready))
  = allRSInStates(SetRS,initial) .
```

Step 5-2: Begin with the most general case.
The most general case is as follows where `sRS` and `sPR` are arbitrary constants for a set of resources and properties respectively:

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

The instantiated proved lemma is effective enough to prove the most general goal.

Step 6-0: Define a predicate to be proved.
Using the double negation idiom, predicate `invinv` for condition (6) 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-ifRSStartedThenPRReady(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 as described in Section 6.1.

Step 6-1: Instantiate proved lemmas for predefined predicates.
Rule `R02` increases `ready` properties which has intuitively no effect on `inv-ifRSStartedThenPRReady`. As explained in Section 5.3.2, proved lemma `m2o-lemma11` ensures it and can be used for proof of condition (6) as follows:

```
vars IDRS IDRRS : RSID
var IDPR : PRID
var TPR : PRType
var SetRS : SetOfResource
var SetPR : SetOfProperty
-- Instantiating m2o-lemma11:
-- 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-lemma11]:
   (ifRSInStatesThenPRInStates
```

```
 (SetRS,started,(prop(TPR,IDPR,notready,IDRS,IDRRS) SetPR),ready)
and
 ifRSInStatesThenPRInStates
 (SetRS,started,(prop(TPR,IDPR,   ready,IDRS,IDRRS) SetPR),ready))
=
 ifRSInStatesThenPRInStates
 (SetRS,started,(prop(TPR,IDPR,notready,IDRS,IDRRS) SetPR),ready) .
```

Step 6-2: Begin with the cases each of which matches to LHS of each rule.
Step 6-3: Split the current case for a rule into cases where the condition of the rule does or does not hold.

We can begin with two cases for two rules. Since rule `R01` is conditional, the general case should be split into two cases according to Step 6-3. The first case should be split into more two cases where the set of properties is or is not empty. Then, the goals of totally four cases can be proved and thus condition (6) is proved as follows:

```
-- Goal of Condition (6) for rule R01
:goal {
  eq invinv(< (res(trs,idRS,initial) sRS), sPR >) = true .
}
:ctf {
  eq allPROfRSInStates(sPR,idRS,ready) = true .
}
-- Case 1: When the condition of R01 holds:
:ctf {
  eq sPR = empPR .
}
-- Case 1-1: sPR is empty.
:apply (rd) -- 1-1
-- Case 1-2: sPR is not empty.
:apply (rd) -- 1-2
-- Case 2: When the condition of R01 does not hold:
:apply (rd) -- 2

-- Goal of Condition (6) for rule R02
:goal {
  eq invinv(< (res(trs,idRRS,started) sRS),
              (prop(tpr,idPR,notready,idRS,idRRS) sPR) >) = true .
}
:apply (rd) -- goal
```

Proof of Invariants for Structural Constraints

We should prove all wfs predicates as invariants, however, they are included in *init* and so we only need to prove condition (6) for each of them. Most of them check some structural constraints of the cloud systems, which should usually keep to hold when some transition rule only changes a local state and does not change any links of some object. When a wfs predicate is defined using predefined predicates, it is easy to prove the invariant property of the wfs because the framework provides many lemmas for the predefined predicates.

Here we use `wfs-allPRHaveRS` as an example to show the procedure.

Step 6-0: Define a predicate to be proved.

```
var S : State
eq invK(S) = wfs-allPRHaveRS(S) .
```

Step 6-1: Instantiate proved lemmas for predefined predicates.
Since `wfs-allPRHaveRS` uses the predefined predicate `allZHaveX`, the proved lemma `m2o-lemma05` can be instantiated and used for proof of condition (6) as follows:

```
var IDRS : RSID
var TPR : PRType
var SetRS : SetOfResource
var SetPR : SetOfProperty
-- Instantiating m2o-lemma05:
--  eq m2o-lemma05(X,X',S_Z,S_X)
--     = allZHaveX(S_Z,(X S_X)) implies allZHaveX(S_Z,(X' S_X))
--     when id(X) = id(X') .
eq [m2o-lemma05]:
   (allPRHaveRS(SetPR,(res(TRS,IDRS,initial) SetRS))
    and allPRHaveRS(SetPR,(res(TRS,IDRS,started) SetRS)))
  = allPRHaveRS(SetPR,(res(TRS,IDRS,initial) SetRS)) .
```

Step 6-2: Begin with the cases each of which matches to LHS of each rule.
Step 6-3: Split the current case for a rule into cases where the condition of the rule does or does not hold.

```
-- Goal of Condition (6) for rule R01
:goal {
  eq invinv(< (res(trs,idRS,initial) sRS), sPR >) = true .
}
:ctf {
  eq allPROfRSInStates(sPR,idRS,ready) = true .
}
-- Case 1: When the condition of R01 holds:
:apply (rd) -- 1
-- Case 2: When the condition of R01 does not hold:
:apply (rd) -- 2

-- Goal of Condition (6) for rule R02
:goal {
  eq invinv(< (res(trs,idRRS,started) sRS),
              (prop(tpr,idPR,notready,idRS,idRRS) sPR) >) = true .
}
:apply (rd) -- goal
```

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 (6) for $noCycle_C$. The Depending Subset Lemma described in Section 5.3 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 (6).

Step 6-0: 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 6-1: Instantiate proved lemmas for predefined predicates.
As described in Section 5.3, the proved lemma `m2o-lemma24` and `set-lemma12` can be instantiated and used for proof as follows:

```
var S : State
vars RS RS' : Resource
var SetRS : SetOfResource
var SetSRS : SetOfRSState
var SetPR : SetOfProperty

-- Instantiating set-lemma12:
-- eq set-lemma12(S) = subset(S,S) .
eq [set-lemma12]:
  subset(SetRS,SetRS) = true .

-- Instantiating m2o-lemma24:
-- 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 id(X) = id(X') and not state(X') \in St_X .
ceq [m2o-lemma24]:
  subset(getRRSsOfPRsInStates((RS' SetRS),SetPR,SetSRS),
         getRRSsOfPRsInStates((RS  SetRS),SetPR,SetSRS))
  = true
  if id(RS) = id(RS') and not state(RS') \in SetSRS .
```

Step 6-2: Begin with the cases each of which matches to LHS of each rule.
Step 6-3: Split the current case for a rule into cases where the condition of the rule does or does not hold.

The following is a proof score for $DDS_C(X,S') \subseteq DDS_C(X,S)$ for rule `R01`:

```
:goal {
  eq invnoRSCycle(x,< (res(trs,idRS,initial) sRS), sPR >) = true .
}
:ctf {
  eq allPROfRSInStates(sPR,idRS,ready) = true .
}
-- Case 1: When the condition of R01 holds:
:ctf {
```

```
  eq x = res(trs,idRS,initial) .
}
-- Case 1-1: X is the resource with identifier idRS.
:apply (rd) -- 1-1
-- Case 1-2: X is not the resource with identifier idRS.
:ctf {
  eq state(x) = initial .
}
-- Case 1-2-1: The resource is initial.
:apply (rd) -- 1-2-1
-- Case 1-2-2: The resource is not initial.
:apply (rd) -- 1-2-2
-- Case 2: When the condition of R01 does not hold:
:apply (rd) -- 2
```

Additionally two `:ctf` commands are required to split Case 1 into three more cases since the considering global state explicitly includes a resource with identifier `idRS`. The three cases are where the considering resource `x` is the resource `idRS` (Case 1-1), is not `idRS` and is `initial` (Case 1-2-1), and is neither `idRS` nor `initial` (Case 1-2-2).

The following is a proof score for $DDS_C(X, S') \subseteq DDS_C(X, S)$ for rule `R02`:

```
:goal {
  eq invnoRSCycle(x,
        < (res(trs,idRRS,started) sRS),
          (prop(tpr,idPR,notready,idRS,idRRS) sPR) >) = true .
}
:ctf {
  eq x = res(trs,idRRS,started) .
}
-- Case 1: X is the resource with identifier idRRS.
:apply (rd) -- 1
-- Case 2: X is not the resource with identifier idRRS.
:ctf {
  eq state(x) = initial .
}
-- Case 2-1: The resource is initial.
:ctf {
  eq id(x) = idRS .
}
-- Case 2-1-1: The identifier of X is idRS.
:apply (rd) -- 2-1-1
-- Case 2-1-2: The identifier of X is not idRS.
:apply (rd) -- 2-1-2
-- Case 2-2: The resource is not initial.
:apply (rd) -- 2-2
```

Similarly, additional case splitting is required since the considering global state includes two identifiers of resources. We need to consider cases where `x` is or is not `idRS` or `idRRS`.

6.7 A Lemma for Using Cyclic Dependency Lemma

Let us return to proof of condition (2) for rule R02.

Step 2-1: Begin with the cases each of which matches to LHS of each rule.

```
-- Goal of Condition (2) for rule R02
:goal {
  eq contcont(< (res(trs,idRRS,started) sRS),
                (prop(tpr,idPR,notready,idRS,idRRS) 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.
Rule R02 is unconditional.
Step 2-3: Split the current case into cases where predicate *final* does or does not hold in the next state.
If all of the other resources are started, the next state is final. But it is not the case because we know a notready property has an initial parent resource.
Step 2-7: When there is a dangling link, split the current case into cases where the linked object does or does not exist.

```
-- The parent resource of the property does or does not exist.
:csp {
  eq existRS(sRS,idRS) = false .
  eq sRS = (res(trs',idRS,srs') sRS') .
}
-- Case 1: When the parent resource of the property does not exist:
:apply (rd) -- 1
-- Case 2: When the parent resource of the property exists:
-- The parent resource is initial or started.
:csp {
  eq srs' = initial .
  eq srs' = started .
}
-- Case 2-1: When the parent resource is initial:
... -- More consideration needed.
-- Case 2-2: When the parent resource is started:
:apply (rd) -- 2-2
```

Case 2-1 for rule R02 is the same situation as Case 1-2 for R01 where there is an initial resource in the next state and the Cyclic Dependency Lemma ensures there exists some initial resource RS such that no resource in DDSC(RS,S) is initial. Thus, here we need to write almost the same proof score as of Case 1-2 for R01. In addition, since we choose an arbitrary initial resource in Case 1-2 for R01, we can assume itself is the resource RS which the Cyclic Dependency Lemma ensures to exist. In this case, however, the initial resource we have is a parent of the property which rule R02 make transit. It means that we should consider two similar cases where the resource RS is the parent resource or another arbitrary resource. We might have to repeat almost the same proof totally three times.

Thereby, it is wise to define a lemma and use it in the similar cases. The lemma claims that if there is an initial resource in a global state then there exists a transition rule applicable to the global state. It can be proved very similar to the proof of condition 1 as follows.

Step 1-0: Define a predicate to be proved.

```
vars B1 B2 : Bool

pred (_when _) : Bool Bool { prec: 64 r-assoc }
eq (B1 when B2)
   = B2 implies B1 .

var S: State

pred invcont : State
eq invcont(S)
  = cont(S) = true
  when inv(S) .
```

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

```
:goal {eq invcont(< (res(trs, idRS, initial) sRS), sPR >) = true .}
```

Step 1-2: Think which rule is applied to the global state in the current case. The rule is referred to as the *current rule*.
The applicable transition may be `R01` because the global state includes an `initial` resource.

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.
The global state already matches to LHS of `R01`.

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

```
:csp {
  eq allPROfRSInStates(sPR,idRS,ready) = true .
  eq sPR = (prop(tpr,idPR,notready,idRS,idRRS) sPR') .
}
-- Case 1: When all of or properties of the resource idRS are ready:
:apply (rd) -- 1
```

Step 1-5: When there is a dangling link, split the current case into cases where the linked object does or does not exist.

```
-- Case 2: When at least one of properties of the resource idRS is notready.
-- The resource referred by the property does or does not exist.
:csp {
  eq existRS(sRS,idRRS) = false .
  eq sRS = (res(trs',idRRS,srs) sRS') .
}
-- Case 2-1: When the resource referred by the property does not exist:
:apply (rd) -- 2-1
-- Case 2-2: When the resource referred by the property exists:
```

Step 1-2: Think which rule is applied to the global state in the current case.
In this case, the transition rule to be applied may be `R02` because the global state includes a property.

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.

```
-- The state of the resource is initial or started.
:csp {
  eq srs = initial .
  eq srs = started .
}
```

Step 1-6: When falling in a cyclic situation, use the Cyclic Dependency Lemma.

```
-- Case 2-2-1: When the resource idRRS is initial:
-- The Cyclic Dependency Lemma rejects this case.
:init [Cycle] by {
  T:RSType <- trs;
  IDRS:RSID <- idRS;
  S:State <- < (res(trs,idRS,initial) sRS), sPR >;
}
:apply (rd) -- 2-2-1
-- Case 2-2-2: When the resource idRRS is started:
:apply (rd) -- 2-2-2
```

Thus, all cases are successfully proved and we can assume that $cont(S)$ holds for any global state S which include an `initial` resource.

Assuming that *inv* holds, this lemma can be used as follows:

```
var IDRS : RSID
var TRS : RSType
var SetRS : SetOfResource
var SetPR : SetOfProperty
eq cont(< (res(TRS,IDRS,initial) SetRS), SetPR >) true .
```

Then, the proof of the sufficient condition (2) for rule `R02` becomes very simple as follow:

```
-- Goal of Condition (2) for rule R02
:goal {
  eq contcont(< (res(trs,idRRS,started) sRS),
                (prop(tpr,idPR,notready,idRS,idRRS) sPR) >) = true .
}
-- The parent resource of the property does or does not exist.
:csp {
  eq existRS(sRS,idRS) = false .
  eq sRS = (res(trs',idRS,srs') sRS') .
}
-- Case 1: When the parent resource of the property does not exist:
:apply (rd) -- 1
-- Case 2: When the parent resource of the property exists:
-- The parent resource is initial or started.
:csp {
  eq srs' = initial .
  eq srs' = started .
}
```

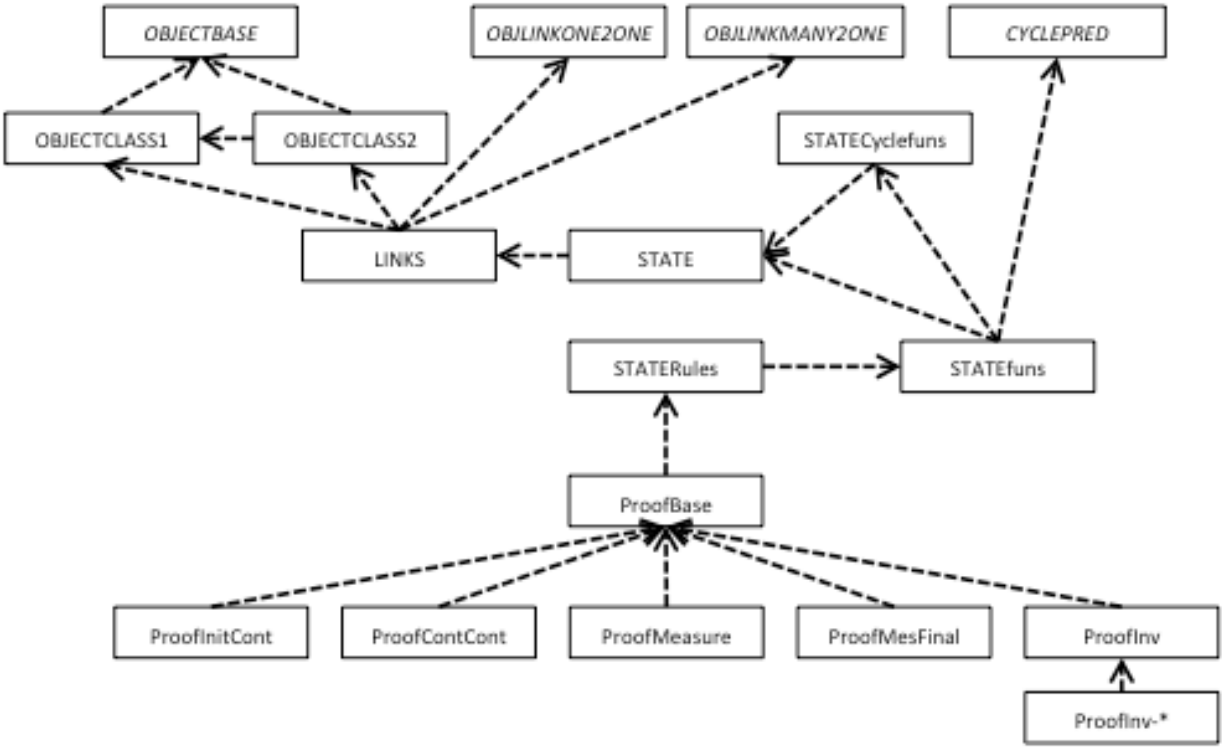

Figure 6.3: Recommended Module Structure

```
-- Case 2-1: When the parent resource is initial:
:apply (rd) -- 2-1
-- Case 2-2: When the parent resource is started:
:apply (rd) -- 2-2
```

6.8 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 6.3 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.

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 `noCycle`.
  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 six 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 arbitrary constants to use in the verification.

- `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.
- `ProofMesFinal`
  Module for proving sufficient condition (4).
  1. Protecting import `ProofBase`.
  2. Define predicate `mesfinal`.
  3. Define axioms of `Nat`.
  4. Define problem specific lemmas if any.
- `ProofInv`
  Module for common definitions for proving sufficient condition (5) and (6).
  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.

Chapter 7

Applying the Framework to TOSCA Specifications

This chapter describes how to use our framework to define behavior of TOSCA types and to verify that a specified topology can correctly automate to set up the cloud system.

7.1 Structure Model of TOSCA Templates

A TOSCA topology models a cloud system that it consists of four classes of objects corresponding to the four main kinds of elements of a topology, i.e. nodes, relationships, capabilities, and requirements. There is an additional object, a message pool, to represent messaging between resources inside of different VMs because they cannot communicate directly. The message pool is simply a bag of messages, which abstracts implementations of messaging.

There are several domain specific constraints of the structure:

1. A node should be hosted on at most one other node.
2. A relationship should not be between a capability and a requirement of the same node.
3. A local relationship should be between a capability and a requirement of the nodes hosted on the same virtual machine.
4. A remote relationship should be between a capability and a requirement of the nodes hosted on the different virtual machines.
5. We assume that types of capabilities and requirements are the same as relationships that link them in this paper for the sake of simplicity.

7.1.1 Representation of the Example Structure Model

Let us use a typical example where four node types and three relationship types in Fig. 2.6 participate in automation of a setup operation. There are nine nodes of four types, nine capabilities, nine requirements, and nine relationships of three types. An initial global state may be represented in CafeOBJ as the following ground term:

```
< ( node(VM, VMApache, initial)
    node(OS, OSApache, initial)
```

```
  node(MW, ApacheWebServer, initial)
  node(SC, CRMApp, initial)
  node(SC, PhpModule, initial)
  node(VM, VMMySQL, initial)
  node(OS, OSMySQL, initial)
  node(MW, MySQL, initial)
  node(SC, CRMDB, initial) ),
( cap(hostedOn, VMApacheOS, closed, VMApache)
  cap(hostedOn, OSApacheSoftware, closed, OSApache)
  cap(hostedOn, ApacheWebServerWebapps, closed, ApacheWebServer)
  cap(hostedOn, ApacheWebServerModules, closed, ApacheWebServer)
  cap(dependsOn, PhpModulePhpApps, closed, PhpModule)
  cap(hostedOn, VMMySQLOS, closed, VMMySQL)
  cap(hostedOn, OSMySQLSoftware, closed, OSMySQL)
  cap(hostedOn, MySQLDatabases, closed, MySQL)
  cap(connectsTo, CRMDBClients, closed, CRMDB) ),
( req(hostedOn, OSApacheContainer, unbound, OSApache)
  req(hostedOn, ApacheWebServerContainer, unbound, ApacheWebServer)
  req(dependsOn, CRMAppPhpRuntime, unbound, CRMApp)
  req(connectsTo, CRMAppDatabase, unbound, CRMApp)
  req(hostedOn, CRMAppContainer, unbound, CRMApp)
  req(hostedOn, PhpModuleContainer, unbound, PhpModule)
  req(hostedOn, OSMySQLContainer, unbound, OSMySQL)
  req(hostedOn, MySQLContainer, unbound, MySQL)
  req(hostedOn, CRMDBContainer, unbound, CRMDB) ),
( rel(hostedOn, OSApacheHostedOnVMApache,
      VMApacheOS, OSApacheContainer)
  rel(hostedOn, ApacheHostedOnOSApache,
      OSApacheSoftware, ApacheWebServerContainer)
  rel(hostedOn, CRMAppHostedOnApache,
      ApacheWebServerWebapps, CRMAppContainer)
  rel(hostedOn, PhpModuleHostedOnApache,
      ApacheWebServerModules, PhpModuleContainer)
  rel(dependsOn, CRMAppDependsOnPhpModule,
      PhpModulePhpApps, CRMAppPhpRuntime)
  rel(hostedOn, OSMySQLHostedOnVMMySQL,
      VMMySQLOS, OSMySQLContainer)
  rel(hostedOn, MySQLHostedOnOSMySQL,
      OSMySQLSoftware, MySQLContainer)
  rel(hostedOn, CRMDBHostedOnMySQL,
      MySQLDatabases, CRMDBContainer)
  rel(connectsTo, CRMAppConnectsToCRMDB,
      CRMDBClients, CRMAppDatabase) ),
 empMsg >
```

The constructor name represents the class of the object (`node`, `cap`, `req`, `rel`), the first argument is its type (`VM`, `hostedOn`, and so on), the second is its identifier (`VMApache`, `VMApacheOS`, and so on), and the third is its local state. The fourth argument of the capability or requirement object represents a link to its parent. The fourth and fifth arguments of the relationship object represent links to its corresponding capability and requirement respectively. The last term,

empMsg, represents an empty message pool.

The representation of these four classes can be easily defined using the template module OBJECTBASE provided by the framework. Module NODE for the node class is as follows:

```
module! NODE {
    -- Instantiation of Template
    extending(OBJECTBASE
        * {sort ObjIDLt -> NDIDLt,
           sort ObjID -> NDID,
           sort ObjTypeLt -> NDTypeLt,
           sort ObjType -> NDType,
           sort ObjStateLt -> NDStateLt,
           sort ObjState -> NDState,
           sort Object -> Node,
           sort SetOfObject -> SetOfNode,
           sort SetOfObjState -> SetOfNDState,
           op empObj -> empND,
           op empState -> empSND,
           op existObj -> existND,
           op existObjInStates -> existNDInStates,
           op uniqObj -> uniqND,
           op #ObjInStates -> #NodeInStates,
           op getObject -> getNode,
           op allObjInStates -> allNDInStates,
           op allObjOfTypeInStates -> allNDOfTypeInStates,
           op allObjNotInStates -> allNDNotInStates,
           op someObjInStates -> someNDInStates})

  -- Constructor
  -- node(NDType, NDID, NDState) is a Node.
  op node : NDType NDID NDState -> Node {constr}

  -- There are four typical node types.
  ops VM OS MW SC : -> NDTypeLt {constr}

  -- Variables
  var TND : NDType
  var IDND : NDID
  var SND : NDState

  -- Selectors
  eq type(node(TND,IDND,SND)) = TND .
  eq id(node(TND,IDND,SND)) = IDND .
  eq state(node(TND,IDND,SND)) = SND .

  -- Local States
  ops initial created started : -> NDStateLt {constr}
  -- Predicate for Local States
  pred isCreated : NDState
  eq isCreated(initial) = false .
  eq isCreated(created) = true .
```

```
  eq isCreated(started) = true .
}
```

The types of nodes are VM (virtual machine), OS (operating system), MW (middleware), and SC (software component). The local states of nodes are `initial`, `created`, and `started`. Among them, `created` and `started` are `isCreated`.

In addition to the predefined predicates/operators explained in Section 5.2, module NODE instantiates a predicate concerning the node types, `allObjOfTypeInStates`, described as follows whereas argument *seto* is a set of linking objects, *setls* is a set of local states of linking objects, and *ty* is a type of an object:

- `allObjOfTypeInStates` (renamed as `allNDOfTypeInStates`)
  Predicate used as `allObjOfTypeInStates`(*seto*,*ty*,*setls*) which holds iff every object of type *ty* in *seto* is in one of local states of *setls*;
  $\forall o \in seto : \texttt{type}(o) = ty \rightarrow \texttt{state}(o) \in setls$.

Since a capability links to its parent node, module CAPABILITY for its class protecting includes NODE as follows:

```
module! CAPABILITY {
  protecting(NODE)

  -- Instantiation of Template
  extending(OBJECTBASE
        * {sort ObjIDLt -> CPIDLt,
           sort ObjID -> CPID,
           sort ObjTypeLt -> CPTypeLt,
           sort ObjType -> CPType,
           sort ObjStateLt -> CPStateLt,
           sort ObjState -> CPState,
           sort Object -> Capability
           sort SetOfObject -> SetOfCapability,
           sort SetOfObjState -> SetOfCPState,
           op empObj -> empCP,
           op empState -> empSCP,
           op existObj -> existCP,
           op existObjInStates -> existCPInStates,
           op uniqObj -> uniqCP,
           op #ObjInStates -> #CapabilityInStates,
           op getObject -> getCapability,
           op allObjInStates -> allCPInStates,
           op allObjOfTypeInStates -> allCPOfTypeInStates,
           op allObjNotInStates -> allCPNotInStates,
           op someObjInStates -> someCPInStates})

  -- Constructor
  -- cap(CPType, CPID, CPState, NDID) is a Capability of a Node
  op cap : CPType CPID CPState NDID -> Capability {constr}

  -- Variables
  var TCP : CPType
```

```
  var IDCP : CPID
  var SCP : CPState
  var IDND : NDID

  -- Selectors
  op node : Capability -> NDID
  eq type(cap(TCP,IDCP,SCP,IDND)) = TCP .
  eq id(cap(TCP,IDCP,SCP,IDND)) = IDCP .
  eq state(cap(TCP,IDCP,SCP,IDND)) = SCP .
  eq node(cap(TCP,IDCP,SCP,IDND)) = IDND .

  -- Local States
  ops closed open available : -> CPStateLt {constr}
  -- Predicate for Local States
  pred isActivated : CPState
  eq isActivated(closed) = false .
  eq isActivated(open) = true .
  eq isActivated(available) = true .
}
```

Note that node is a selector for a link to the parent node of the capability. The local states of capabilities are closed, open, and available. Among them, open and available are isActivated.

Since a requirement also links to its parent node, module REQUIREMENT for its class protecting includes NODE as follows:

```
module! REQUIREMENT {
  protecting(NODE)

  -- Instantiation of Template
  extending(OBJECTBASE
        * {sort ObjIDLt -> RQIDLt,
           sort ObjID -> RQID,
           sort ObjTypeLt -> RQTypeLt,
           sort ObjType -> RQType,
           sort ObjStateLt -> RQStateLt,
           sort ObjState -> RQState,
           sort Object -> Requirement,
           sort SetOfObject -> SetOfRequirement,
           sort SetOfObjState -> SetOfRQState,
           op empObj -> empRQ,
           op empState -> empSRQ,
           op existObj -> existRQ,
           op existObjInStates -> existRQInStates,
           op uniqObj -> uniqRQ,
           op #ObjInStates -> #RequirementInStates,
           op getObject -> getRequirement,
           op allObjInStates -> allRQInStates,
           op allObjOfTypeInStates -> allRQOfTypeInStates,
           op allObjNotInStates -> allRQNotInStates,
           op someObjInStates -> someRQInStates})
```

```
    -- Constructor
    -- req(RQType, RQID, RQState, NDID) is a Requirement of a Node
    op req : RQType RQID RQState NDID -> Requirement {constr}

    -- Variables
    var TRQ : RQType
    var IDRQ : RQID
    var IDND : NDID
    var SRQ : RQState

    -- Selectors
    op node : Requirement -> NDID
    eq type(req(TRQ,IDRQ,SRQ,IDND)) = TRQ .
    eq id(req(TRQ,IDRQ,SRQ,IDND)) = IDRQ .
    eq state(req(TRQ,IDRQ,SRQ,IDND)) = SRQ .
    eq node(req(TRQ,IDRQ,SRQ,IDND)) = IDND .

    -- Local States
    ops unbound waiting ready : -> RQStateLt {constr}
}
```

Note that node is a selector for a link to the parent node of the requirement. The local states of requirements are unbound, waiting, and ready.

Since a relationship links to its corresponding capability and requirement, module RELATIONSHIP for its class protecting includes CAPABILITY and REQUIREMENT as follows:

```
  module! RELATIONSHIP {
    protecting(CAPABILITY + REQUIREMENT)

    -- Instantiation of Template
    extending(OBJECTBASE
          * {sort ObjIDLt -> RLIDLt,
             sort ObjID -> RLID,
             sort ObjTypeLt -> RLTypeLt,
             sort ObjType -> RLType,
             sort ObjStateLt -> RLStateLt,
             sort ObjState -> RLState,
             sort Object -> Relationship,
             sort SetOfObject -> SetOfRelationship,
             sort SetOfObjState -> SetOfRLState,
             op empObj -> empRL,
             op existObj -> existRL,
             op uniqObj -> uniqRL})

    -- Constructor
    -- rel(RLType, RLID, CPID, RQID) is a Relationship
    op rel : RLType RLID CPID RQID -> Relationship {constr}

    -- There are three typical relationship types.
    ops hostedOn dependsOn connectsTo : -> RLTypeLt {constr}
```

```
  -- Types of capabilities and requirements are the same as relationships
  [RLType < CPType RQType]

  -- Variables
  var TRL : RLType
  var IDRL : RLID
  var IDCP : CPID
  var IDRQ : RQID

  -- Selectors
  op cap : Relationship -> CPID
  op req : Relationship -> RQID
  eq type(rel(TRL,IDRL,IDCP,IDRQ)) = TRL .
  eq id(rel(TRL,IDRL,IDCP,IDRQ)) = IDRL .
  eq cap(rel(TRL,IDRL,IDCP,IDRQ)) = IDCP .
  eq req(rel(TRL,IDRL,IDCP,IDRQ)) = IDRQ .

  -- Predicate for Locality
  pred isLocalRL : Relationship
  eq isLocalRL(rel(hostedOn,IDRL,IDCP,IDRQ)) = true .
  eq isLocalRL(rel(dependsOn,IDRL,IDCP,IDRQ)) = true .
  eq isLocalRL(rel(connectsTo,IDRL,IDCP,IDRQ)) = false .
}
```

Sort RLType is declared as a subsort of CPType and RQType which means types of relationships can be used as types of capabilities and requirements. The types of relationships are hostedOn, dependsOn, and connectsTo. Among them,
hostedOn and dependsOn are isLocal. Note that cap and req are selectors for links to the corresponding capability and requirement respectively of the relationship.

Predefined predicates and operators for links between objects also can be easily instantiated using the template modules OBJLINKMANY2ONE and OBJLINKONE2ONE as follows:

```
module! LINKS {
  protecting(NODE + CAPABILITY + REQUIREMENT + RELATIONSHIP)

  -- Instantiation of Template
  -- A many-to-one link from a capability to its parent node
  extending(OBJLINKMANY2ONE(
        CAPABILITY {sort Object -> Capability,
                    sort ObjID -> CPID,
                    sort ObjType -> CPType,
                    sort ObjState -> CPState,
                    sort SetOfObject -> SetOfCapability,
                    sort SetOfObjState -> SetOfCPState,
                    sort LObject -> Node,
                    sort LObjID -> NDID,
                    sort LObjState -> NDState,
                    sort SetOfLObject -> SetOfNode,
                    sort SetOfLObjState -> SetOfNDState,
                    op getLObject -> getNode,
                    op existLObj -> existND,
```

```
                op empLObj -> empND,
                op link -> node,
                op existLObjInStates -> existNDInStates})
      * {op getXOfZ -> getNDOfCP,
         op getZsOfX -> getCPsOfND,
         op getZsOfTypeOfX -> getCPsOfTypeOfND,
         op getZsOfXInStates -> getCPsOfNDInStates,
         op getZsOfTypeOfXInStates -> getCPsOfTypeOfNDInStates,
         op getXsOfZs -> getNDsOfCPs,
         op getXsOfZsInStates -> getNDsOfCPsInStates,
         op getZsOfXs -> getCPsOfNDs,
         op getZsOfXsInStates -> getCPsOfNDsInStates,
         op getZsOfTypeOfXsInStates -> getCPsOfTypeOfNDsInStates,
         op allZHaveX -> allCPHaveND,
         op allZOfXInStates -> allCPOfNDInStates,
         op allZOfTypeOfXInStates -> allCPOfTypeOfNDInStates,
         op ifXInStatesThenZInStates -> ifNDInStatesThenCPInStates,
         op ifXInStatesThenZOfTypeInStates
            -> ifNDInStatesThenCPOfTypeInStates}
      )

-- Instantiation of Template
-- A many-to-one link from a requirement to its parent node
extending(OBJLINKMANY2ONE(
      REQUIREMENT {sort Object -> Requirement,
                   sort ObjID -> RQID,
                   sort ObjType -> RQType,
                   sort ObjState -> RQState,
                   sort SetOfObject -> SetOfRequirement,
                   sort SetOfObjState -> SetOfRQState,
                   sort LObject -> Node,
                   sort LObjID -> NDID,
                   sort LObjState -> NDState,
                   sort SetOfLObject -> SetOfNode,
                   sort SetOfLObjState -> SetOfNDState,
                   op getLObject -> getNode,
                   op existLObj -> existND,
                   op empLObj -> empND,
                   op link -> node,
                   op existLObjInStates -> existNDInStates})
      * {op getXOfZ -> getNDOfRQ,
         op getXsOfZs -> getNDsOfRQs,
         op getXsOfZsInStates -> getNDsOfRQsInStates,
         op getZsOfX -> getRQsOfND,
         op getZsOfTypeOfX -> getRQsOfTypeOfND,
         op getZsOfXInStates -> getRQsOfNDInStates,
         op getZsOfTypeOfXInStates -> getRQsOfTypeOfNDInStates,
         op getZsOfXs -> getRQsOfNDs,
         op getZsOfXsInStates -> getRQsOfNDsInStates,
         op getZsOfTypeOfXsInStates -> getRQsOfTypeOfNDsInStates,
```

```
         op allZHaveX -> allRQHaveND,
         op allZOfXInStates -> allRQOfNDInStates,
         op allZOfTypeOfXInStates -> allRQOfTypeOfNDInStates,
         op ifXInStatesThenZInStates -> ifNDInStatesThenRQInStates,
         op ifXInStatesThenZOfTypeInStates
            -> ifNDInStatesThenRQOfTypeInStates}
      )

-- Instantiation of Template
-- A one-to-one link from a relationship to its capability
extending(OBJLINKONE2ONE(
      RELATIONSHIP {sort Object -> Relationship,
                    sort ObjID -> RLID,
                    sort ObjType -> RLType,
                    sort ObjState -> RLState,
                    sort SetOfObject -> SetOfRelationship,
                    sort SetOfObjState -> SetOfRLState,
                    sort LObject -> Capability,
                    sort LObjID -> CPID,
                    sort LObjState -> CPState,
                    sort SetOfLObject -> SetOfCapability,
                    sort SetOfLObjState -> SetOfCPState,
                    op getLObject -> getCapability,
                    op existLObj -> existCP,
                    op empLObj -> empCP,
                    op link -> cap,
                    op existLObjInStates -> existCPInStates})
      * {op existX -> existCP,
         op getXOfY -> getCPOfRL,
         op getXsOfYs -> getCPsOfRLs,
         op getXsOfYsInStates -> getCPsOfRLsInStates,
         op getYOfX -> getRLOfCP,
         op getYsOfXs -> getRLsOfCPs,
         op getYsOfXsInStates -> getRLsOfCPsInStates,
         op uniqX -> uniqCP,
         op YOfXInStates -> RLOfCPInStates,
         op ifXInStatesThenYInStates -> ifCPInStatesThenRLInStates,
         op ifYInStatesThenXInStates -> ifRLInStatesThenCPInStates,
         op allYHaveX -> allRLHaveCP,
         op allXHaveY -> allCPHaveRL,
         op onlyOneYOfX -> onlyOneRLOfCP}
      )

-- Instantiation of Template
-- A one-to-one link from a relationship to its relationship
extending(OBJLINKONE2ONE(
      RELATIONSHIP {sort Object -> Relationship,
                    sort ObjID -> RLID,
                    sort ObjType -> RLType,
                    sort ObjState -> RLState,
```

```
                    sort SetOfObject -> SetOfRelationship,
                    sort SetOfObjState -> SetOfRLState,
                    sort LObject -> Requirement,
                    sort LObjID -> RQID,
                    sort LObjState -> RQState,
                    sort SetOfLObject -> SetOfRequirement,
                    sort SetOfLObjState -> SetOfRQState,
                    op getLObject -> getRequirement,
                    op existLObj -> existRQ,
                    op empLObj -> empRQ,
                    op link -> req,
                    op existLObjInStates -> existRQInStates})
        * {op existX -> existRQ,
           op getXOfY -> getRQOfRL,
           op getXsOfYs -> getRQsOfRLs,
           op getXsOfYsInStates -> getRQsOfRLsInStates,
           op getYOfX -> getRLOfRQ,
           op getYsOfXs -> getRLsOfRQs,
           op getYsOfXsInStates -> getRLsOfRQsInStates,
           op uniqX -> uniqRQ,
           op YOfXInStates -> RLOfRQInStates,
           op ifXInStatesThenYInStates -> ifRQInStatesThenRLInStates,
           op ifYInStatesThenXInStates -> ifRLInStatesThenRQInStates,
           op allYHaveX -> allRLHaveRQ,
           op allXHaveY -> allRQHaveRL,
           op onlyOneYOfX -> onlyOneRLOfRQ}
        )
}
```

Links from capabilities to their parent nodes and from requirements to their parent nodes are many-to-one, whereas links from relationships to their corresponding capabilities and requirements are one-to-one.

In addition to the predefined predicates/operators explained in Section 5.2, module `LINKS` uses `OBJLINKMANY2ONE` to instantiate several operators concerning object types. The following is the list of them whereas argument *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, *setlls* is a set of local states of linked objects, and *ty* is a type of an object:

- `allZOfTypeOfXInStates`
  (renamed as `allCPOfTypeOfNDInStates` and `allRQOfTypeOfNDInStates`)
  Predicate used as `allZOfTypeOfXInStates`(*seto*,*ty*,*lid*,*setls*) which holds iff every object included in *seto* whose type is *ty* and whose link is *lid* is in one of locals state in *setls*;
  $\forall o \in seto : (\texttt{type}(o) = ty \wedge \texttt{link}(o) = lid \rightarrow \texttt{state}(o) \in setls)$.

- `getZsOfTypeOfX` (as `getCPsOfTypeOfND` and `getRQsOfTypeOfND`)
  Operator used as `getZsOfTypeOfX`(*seto*,*ty*,*lobj*) which returns a subset *seto* each of whose element object is of type *ty* and links to *lobj*.

- `getZsOfTypeOfXInStates`
  (as `getCPsOfTypeOfNDInStates` and `getRQsOfTypeOfNDInStates`)

Operator used as `getZsOfTypeOfXInStates(`*seto*`,`*tylobj*`,`*setls*`)` which returns a subset of *seto* each of whose element object is of type *ty*, links to *lobj*, and is in one of local states of *setls*.

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

- `ifXInStatesThenZOfTypeInStates`
  (as `ifNDInStatesThenCPOfTypeInStates` and `ifNDInStatesThenRQOfTypeInStates`)
  Predicate used as `ifXInStatesThenZOfTypeInStates(`*setlo*`,`*ty*`,`*setlls*`,`*seto*`,`*setls*`)` which holds iff every object included in *setlo* whose type m is *ty* and 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*;

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

Module `LINKS` also uses `OBJLINKONE2ONE` to instantiate many predicates/operators. The following is the list of them 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*;
  $\exists o \in seto : \texttt{link}(o) = lid \wedge (\forall o' \in seto : o \neq o' \rightarrow \texttt{link}(o) \neq \texttt{link(o')})$.

A global state of the TOSCA structure models includes one additional object, a message pool. There are two kinds of messages, open messages and available messages, which will be explained in the next section. The representation of the messages is defined as follows:

```
module! MSG {
  protecting(LINKS)
  [Msg]
  -- An open message
```

```
  op opMsg : CPID -> Msg {constr}
  -- An available message
  op avMsg : CPID -> Msg {constr}

  vars IDCP1 IDCP2 : CPID
  eq (opMsg(IDCP1) = opMsg(IDCP2))
     = (IDCP1 = IDCP2) .
  eq (avMsg(IDCP1) = avMsg(IDCP2))
     = (IDCP1 = IDCP2) .
  eq (opMsg(IDCP1) = avMsg(IDCP2))
     = false .
}
```

An open message (and also an available message) has an argument of the identifier of a capability. Open messages are equal to each other iff they have the same capability identifier, which is similar to available messages. An open message and an available message are never equal to each other.

The representation of a global state is defined by sort `State` as a tuple consisting of a set of nodes, a set of capabilities, a set of requirements, a set of relationships, and a message pool as follows whereas parameterized module `BAG` defines generic bags similarly to module `SET` explained in Section 3.1:

```
module! STATE {
  protecting(LINKS)
  protecting(BAG(MSG {sort Elt -> Msg})
      * {sort Bag -> PoolOfMsg,
         op empty -> empMsg})

  [State]
  op <_,_,_,_,_> : SetOfNode SetOfCapability SetOfRequirement
                   SetOfRelationship PoolOfMsg -> State {constr}
}
```

In addition to the instantiated operators from the predefined ones, it requires to define several problem specific operators in module `STATEfuns`. There are three kinds of them; (1) to represent invariants for the consistency between messages and local states of objects, (2) to represent invariants for the consistency between capabilities and requirements connected by relationships, and (3) to represent other problem specific constraints. All these operators can be easily implemented by combining predefined operators.

The following is the list of them whereas argument *setCP* is a set of capabilities, *setRQ* is a set of requirements, *setRL* is a set of relationships, *node* is a node, *cap* is a capability, *req* is a requirement, *rel* is a relationship, *setlCP* is a set of local states of capabilities, *setlRQ* is a set of local states of requirements, and *pool* is a message pool[1]:

- `allHostedOnCPInStates` (categorized as (3) above)
  Predicate used as `allHostedOnCPInStates`(*setCP*, *setlCP*) which holds iff every capability included in *setCP* whose type is `hostedOn` is in one of locals state in *setlCP*;
  $\forall cap \in setCP : (\texttt{type}(cap) = \texttt{hostedOn} \rightarrow \texttt{state}(cap) \in setlCP)$.

[1]Note that here we do not distinguish between an object and its identifier for the sake of brevity.

- `allHostedOnRQInStates` (categorized as (3))
  Predicate used as `allHostedOnRQInStates(`*setRQ*`,`*setlRQ*`)` which holds iff every requirement included in *setRQ* whose type is `hostedOn` is in one of locals state in *setlRQ*;
  $\forall req \in setRQ : (\texttt{type}(req) = \texttt{hostedOn} \rightarrow \texttt{state}(req) \in setlRQ)$.

- `allHostedOnRQOfNDInStates` (categorized as (3))
  Predicate used as `allHostedOnRQOfNDInStates(`*setRQ*`,`*node*`,`*setlRQ*`)` which holds iff every requirement included in *setRQ* whose type is `hostedOn` and whose parent is *node* is in one of locals state in *setlRQ*;
  $\forall req \in setRQ : (\texttt{type}(req) = \texttt{hostedOn} \wedge \texttt{node}(req) = node \rightarrow \texttt{state}(req) \in setlRQ)$.

- `getCPOfRQ` (categorized as (2))
  Operator used as `getCPOfRQ(`*setCP*`,`*setRL*`,`*req*`)` which returns the corresponding capability of *req* by firstly finding the corresponding relationship of *req* in *setRL* and then finding the corresponding capability of the relationship in *setCP*.

- `getRQOfCP` (categorized as (2))
  Operator used as `getRQOfCP(`*setRQ*`,`*setRL*`,`*cap*`)` which returns the corresponding requirement of *cap* by firstly finding the corresponding relationship of *cap* in *setRL* and then finding the corresponding requirement of the relationship in *setRQ*.

- `allRLHaveSameTypeCPRQ` (categorized as (3))
  Predicate used as `allRLHaveSameTypeCPRQ(`*setRL*`,`*setCP*`,`*setRQ*`)` which holds iff every relationship included in *setRL* has the corresponding capability included in *setCP* and the corresponding requirement included in *setRQ* and those three objects has the same type;
  $\forall rel \in setRL :$
  $(\forall cap \in setCP : \texttt{cap}(rel) = cap \rightarrow \texttt{type}(rel) = \texttt{type}(cap)) \wedge$
  $(\forall req \in setRQ : \texttt{req}(rel) = req \rightarrow \texttt{type}(rel) = \texttt{type}(req))$.

- `allRLNotInSameND` (categorized as (3))
  Predicate used as `allRLNotInSameND(`*setRL*`,`*setCP*`,`*setRQ*`)` which holds iff every relationship included in *setRL* has the corresponding capability included in *setCP* and the corresponding requirement included in *setRQ* and their parent nodes are not the same;
  $\forall rel \in setRL, \exists cap \in setCP, \exists req \in setRQ :$
  $\texttt{cap}(rel) = cap \wedge \texttt{req}(rel) = req \wedge \texttt{node}(cap) \neq \texttt{node}(req)$.

- `getHostedOnRQOfND` (categorized as (3))
  Operator used as `getHostedOnRQOfND(`*setRQ*`,`*node*`)` which returns the `hostedOn` requirement in *setRQ* whose parent is *node*.

- `getHostedOnRQsOfNDInStates` (categorized as (3))
  Operator used as `getHostedOnRQsOfNDInStates(`*setRQ*`,`*node*`,`*setlRQ*`)` which returns the set of `hostedOn` requirements in *setRQ* whose parent is *node* and whose local state is in *setlRQ*.

- `VMOfND` (categorized as (3))
  Operator used as `VMOfND(`*node*`,`*setND*`,`*setCP*`,`*setRQ*`,`*setRL*`)` which returns the `VM` node which hosts *node*; precisely, the operator recursively traverses `hostedOn` requirements,

relationships, and capabilities starting from *node* and returns the first found `VM` node including *node* itself.

- `VMOfCP` (categorized as (3))
  Operator used as `VMOfCP`(*cap*, *setND*, *setCP*, *setRQ*, *setRL*) which returns the `VM` node which hosts the parent node of *cap*.

- `VMOfRQ` (categorized as (3))
  Operator used as `VMOfRQ`(*req*, *setND*, *setCP*, *setRQ*, *setRL*) which returns the `VM` node which hosts the parent node of *req*.

- `allRLHoldLocality` (categorized as (3))
  Predicate used as `allRLHoldLocality`(*setRL*, *setND*, *setCP*, *setRQ*) which holds iff every relationship included in *setRL* satisfies the locality constraint, which means that if the type of a relationship is local, it should be between a capability and a requirement of the nodes hosted on the same virtual machine, while if the type is not local (i.e. remote), it should be between a capability and a requirement of the nodes hosted on the different virtual machines.

- `allNDHaveAtMostOneHost` (categorized as (3))
  Predicate used as `allNDHaveAtMostOneHost`(*setND*, *setRQ*) which holds iff every node included in *setND* has 0 or 1 `hostedOn` requirement included in *setRQ*.

- `ifOpenMsgThenCPInStates` (categorized as (1))
  Predicate used as `ifOpenMsgThenCPInStates`(*pool*, *setCP*, *setlCP*) which holds iff every open message included in *pool* has the corresponding capability which is included in *setCP* and whose local state is in *setlCP*;
  $\forall msg \in pool : (isOpen(msg) \rightarrow$
  $(\exists cap \in setCP : cap(msg) = cap \wedge \texttt{state}(cap) \in setlCP))$.

- `ifAvailableMsgThenCPInStates` (categorized as (1))
  Predicate used as `ifAvailableMsgThenCPInStates`(*pool*, *setCP*, *setlCP*) which holds iff every available message included in *pool* has the corresponding capability which is included in *setCP* and whose local state is in *setlCP*;
  $\forall msg \in pool : (isAvail(msg) \rightarrow$
  $(\exists cap \in setCP : cap(msg) = cap \wedge \texttt{state}(cap) \in setlCP))$.

- `ifCPInStatesThenRQInStates` (categorized as (2))
  Predicate used as `ifCPInStatesThenRQInStates`(*setCP*, *setlCP*, *setRQ*, *setlRQ*, *setRL*) which holds iff every capability included in *setCP* whose local sate is included in *setlCP* has the corresponding requirement included in *setRQ* whose local state is in *setlRQ*;
  $\forall cap \in setCP : (\texttt{state}(cap) \in setlCP \rightarrow$
  $\exists rel \in setRL, req \in setRQ :$
  $(\texttt{cap}(rel) = cap \wedge \texttt{req}(rel) = req \wedge \texttt{state}(req) \in setlRQ))$.

- `ifConnectsToCPInStatesThenRQInStatesOrOpenMsg` (categorized as (1) (2))
  Predicate used as `ifConnectsToCPInStatesThenRQInStatesOrOpenMsg`(*setCP*, *setlCP*, *setRQ*, *setlRQ*, *setRL*, *pool*) which holds iff every `connectsTo` capability included in *setCP* whose local sate is included in *setlCP* has the corresponding requirement included in

setRQ whose local state is in *setlRQ* or has the corresponding open message included in *pool*;

$\forall cap \in setCP : (\texttt{type}(cap) = \texttt{hostedOn} \wedge \texttt{state}(cap) \in setlCP \rightarrow$
$(\exists rel \in setRL, req \in setRQ :$
$(\texttt{cap}(rel) = cap \wedge \texttt{req}(rel) = req \wedge \texttt{state}(req) \in setlRQ)) \vee$
$\exists msg \in pool : cap(msg) = cap \wedge isOpen(msg)).$

- `ifConnectsToCPInStatesThenRQInStatesOrAvailableMsg` (categorized as (1) (2))
  Predicate used as `ifConnectsToCPInStatesThenRQInStatesOrAvailableMsg`(*setCP*, *setlCP*, *setRQ*, *setlRQ*, *setRL*, *pool*) which holds iff every `connectsTo` capability included in *setCP* whose local sate is included in *setlCP* has the corresponding requirement included in *setRQ* whose local state is in *setlRQ* or has the corresponding available message included in *pool*;

  $\forall cap \in setCP : (\texttt{type}(cap) = \texttt{hostedOn} \wedge \texttt{state}(cap) \in setlCP \rightarrow$
  $(\exists rel \in setRL, req \in setRQ :$
  $(\texttt{cap}(rel) = cap \wedge \texttt{req}(rel) = req \wedge \texttt{state}(req) \in setlRQ)) \vee$
  $\exists msg \in pool : cap(msg) = cap \wedge isAvail(msg)).$

7.2 Behavior Model of TOSCA Templates

The framework models the behavior of an automated system operation as a state machine in which a set of transition rules of global states specifies the behavior. As described in Section 2.3, the behavior of a topology of TOSCA is decided by the behavior of types of nodes and relationships included in the topology. Here, we propose to model the behavior of a type as a set of transition rules each of which is called an *invocation rule* and specifies when a type operation can be invoked and how it changes the local state of a node or relationship of the type.

As described in Section 2.3, type operations and their invocation rules should be defined by type architects. When an application architect defines a topology, the set of all invocation rules of included node/relationship types collectively composes a state machine which specifies the whole behavior of the topology.

In the example of Fig. 2.6, we assume that behavior of four node types is the same focusing on when a node is created and started because they are the most essential for setup operations.

On the other hand, behavior of relationship types usually varies according to their nature; they may be in the IaaS layer or in the inside of VM layer, “local” or “remote”, “immediate” or “await”. Three relationship types of this example typically cover the variation. A HostedOn relationship is one between resources in the IaaS layer. It is “immediate”, i.e. it can be established as soon as the target node is created. Each of DependsOn and ConnectsTo relationships is between resources inside of VMs and is “await”, i.e. it should wait for the target node to be started. A DependsOn relationship is “local” in the same VM, while a ConnectsTo is “remote” to a different VM and should use some messages to notice the states of its capability to its requirement. We assume that a state of a relationship is a pair of the states of its capability and requirement in this paper for the sake of simplicity. Thereby, an operation of a relationship type changes the state of its capability or requirement.

Behavior of these types is depicted in Fig. 7.1. A solid arrow represents a state transition of each object caused by a type operation and a dashed arrow represents an invocation of a type operation or a message sending.

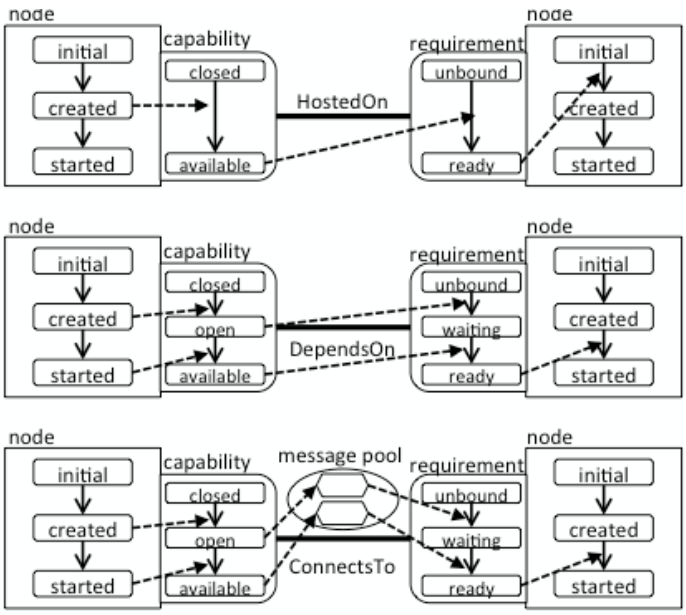

Figure 7.1: Typical Behavior of Relationship Types

There are twelve invocation rules; two of them are for node operations, two are for operations of HostedOn relationship, four are for DependsOn, and four are for ConnectsTo. The followings are detailed definitions of them in English:

Initial States: Every node is initially in a local state named as *initial*, every capability of the node is *closed*, and every requirement is *unbound*.

Invocation Rule of Node Type Operations:

- *CREATE* operation can be invoked if all of the HostedOn requirements of the node are *ready* and changes the local state of the node from *initial* to *created*.
- *START* operation can be invoked if all of the requirements are *ready* and changes the local state from *created* to *started*.

Invocation Rule of Operations of HostedOn Relationship Type:

- *CAPAVAILABLE* operation can be invoked if the target node is already created, i.e. *created* or *started* and changes the local state of its capability from *closed* to *available*.
- *REQREADY* operation can be invoked if its capability is *available* and changes the local state of the requirement from *unbound* to *ready*.

Invocation Rule of Operations DependsOn Relationship Type:

- *CAPOPEN* operation can be invoked if the target node is already created and changes the local state of its capability from *closed* to *open*.
- *CAPAVAILABLE* operation can be invoked if the target node is *started* and changes the local state of its capability from *open* to *available*.

- *REQWAITING* operation can be invoked if its capability is already activated, i.e. *open* or *available*, and the source node is *created*. It changes the local state of its requirement from *unbound* to *waiting*.
- *REQREADY* operation can be invoked if its capability is *available* and changes the local state of its requirement from *waiting* to *ready*.

Invocation Rule Operations of ConnectsTo Relationship Type:

- *CAPOPEN* operation can be invoked if the target node is already created. It changes the local state of its capability from *closed* to *open* and also issues an open message of the capability to the message pool.
- *CAPAVAILABLE* operation can be invoked if the target node is *started*. It changes the local state of its capability from *open* to *available* and also issues an available message of the capability to the message pool.
- *REQWAITING* operation can be invoked if it finds an open message of its capability and the source node is *created*. It changes the local state of its requirement from *unbound* to *waiting*.
- *REQREADY* operation can be invoked if it finds an available message of its capability and changes the local state of its requirement from *waiting* to *ready*.

7.2.1 Representation of the Example Behavior Model

Each of twelve rules explained in English above is more formally represented by a transition rule of CafeOBJ as follows:

```
module! STATERules {
  protecting(STATEfuns)

  -- Variables
  var TND : NDType
  vars IDND IDND1 IDND2 : NDID
  var IDCP : CPID
  var IDRQ : RQID
  var IDRL : RLID
  var SetND : SetOfNode
  var SetCP : SetOfCapability
  var SetRQ : SetOfRequirement
  var SetRL : SetOfRelationship
  var SCP : CPState
  var MP : PoolOfMsg

  -- CREATE Operation for Node Type
  ctrans [R01]:
     < (node(TND,IDND,initial) SetND), SetCP, SetRQ, SetRL, MP >
  => < (node(TND,IDND,created) SetND), SetCP, SetRQ, SetRL, MP >
  if allHostedOnRQOfNDInStates(SetRQ,IDND,ready) .

  -- START Operation for Node Type
```

```
ctrans [R02]:
   < (node(TND,IDND,created) SetND), SetCP, SetRQ, SetRL, MP >
=> < (node(TND,IDND,started) SetND), SetCP, SetRQ, SetRL, MP >
if allRQOfNDInStates(SetRQ,IDND,ready) .

-- CAPAVAILABLE Operation for HostedOn Relationship Type
ctrans [R03]:
   < SetND, (cap(hostedOn,IDCP,closed,   IDND) SetCP), SetRQ, SetRL, MP >
=> < SetND, (cap(hostedOn,IDCP,available,IDND) SetCP), SetRQ, SetRL, MP >
if isCreated(state(getNode(SetND,IDND))) .

-- REQREADY Operation for HostedOn Relationship Type
trans [R04]:
   < SetND, (cap(hostedOn,IDCP,available,IDND1) SetCP),
            (req(hostedOn,IDRQ,unbound,IDND2) SetRQ),
            (rel(hostedOn,IDRL,IDCP,IDRQ) SetRL), MP >
=> < SetND, (cap(hostedOn,IDCP,available,IDND1) SetCP),
            (req(hostedOn,IDRQ,ready,  IDND2) SetRQ),
            (rel(hostedOn,IDRL,IDCP,IDRQ) SetRL), MP > .

-- CAPOPEN Operation for DependsOn Relationship Type
ctrans [R05]:
   < SetND, (cap(dependsOn,IDCP,closed,IDND) SetCP), SetRQ, SetRL, MP >
=> < SetND, (cap(dependsOn,IDCP,open,  IDND) SetCP), SetRQ, SetRL, MP >
if isCreated(state(getNode(SetND,IDND))) .

-- CAPAVAILABLE Operation for DependsOn Relationship Type
ctrans [R06]:
   < SetND, (cap(dependsOn,IDCP,open,     IDND) SetCP), SetRQ, SetRL, MP >
=> < SetND, (cap(dependsOn,IDCP,available,IDND) SetCP), SetRQ, SetRL, MP >
if state(getNode(SetND,IDND)) = started .

-- REQWAITING Operation for DependsOn Relationship Type
ctrans [R07]:
   < SetND, (cap(dependsOn,IDCP,SCP,IDND1) SetCP),
            (req(dependsOn,IDRQ,unbound,IDND2) SetRQ),
            (rel(dependsOn,IDRL,IDCP,IDRQ) SetRL), MP >
=> < SetND, (cap(dependsOn,IDCP,SCP,IDND1) SetCP),
            (req(dependsOn,IDRQ,waiting,IDND2) SetRQ),
            (rel(dependsOn,IDRL,IDCP,IDRQ) SetRL), MP >
if state(getNode(SetND,IDND2)) = created and isActivated(SCP) .

-- REQREADY Operation for DependsOn Relationship Type
trans [R08]:
   < SetND, (cap(dependsOn,IDCP,available,IDND1) SetCP),
            (req(dependsOn,IDRQ,waiting,IDND2) SetRQ),
            (rel(dependsOn,IDRL,IDCP,IDRQ) SetRL), MP >
=> < SetND, (cap(dependsOn,IDCP,available,IDND1) SetCP),
            (req(dependsOn,IDRQ,ready,  IDND2) SetRQ),
            (rel(dependsOn,IDRL,IDCP,IDRQ) SetRL), MP > .
```

```
  -- CAPOPEN Operation for ConnectsTo Relationship Type
  ctrans [R09]:
     < SetND, (cap(connectsTo,IDCP,closed,IDND) SetCP),
       SetRQ, SetRL, MP >
  => < SetND, (cap(connectsTo,IDCP,open,  IDND) SetCP),
       SetRQ, SetRL, (opMsg(IDCP) MP) >
  if isCreated(state(getNode(SetND,IDND))) .

  -- CAPAVAILABLE Operation for ConnectsTo Relationship Type
  ctrans [R10]:
     < SetND, (cap(connectsTo,IDCP,open,     IDND) SetCP),
       SetRQ, SetRL, MP >
  => < SetND, (cap(connectsTo,IDCP,available,IDND) SetCP),
       SetRQ, SetRL, (avMsg(IDCP) MP) >
  if state(getNode(SetND,IDND)) = started .

  -- REQWAITING Operation for ConnectsTo Relationship Type
  ctrans [R11]:
     < SetND, SetCP,
       (req(connectsTo,IDRQ,unbound,IDND) SetRQ),
       (rel(connectsTo,IDRL,IDCP,IDRQ) SetRL),
       (opMsg(IDCP) MP) >
  => < SetND, SetCP,
       (req(connectsTo,IDRQ,waiting,IDND) SetRQ),
       (rel(connectsTo,IDRL,IDCP,IDRQ) SetRL), MP >
  if state(getNode(SetND,IDND)) = created .

  -- REQREADY Operation for ConnectsTo Relationship Type
  trans [R12]:
     < SetND, SetCP,
       (req(connectsTo,IDRQ,waiting,IDND) SetRQ),
       (rel(connectsTo,IDRL,IDCP,IDRQ) SetRL),
       (avMsg(IDCP) MP) >
  => < SetND, SetCP,
       (req(connectsTo,IDRQ,ready,  IDND) SetRQ),
       (rel(connectsTo,IDRL,IDCP,IDRQ) SetRL), MP > .
}
```

7.3 Verification of TOSCA Templates

This section presents the verification of the liveness property of setup operations of the TOSCA models. As described in Chapter 6, reachability of setup operations of cloud systems is formalized as (*init* `leads-to` $final$) and there are six sufficient conditions for it.

7.3.1 Definition of Predicates

Step 0-1: Define *init* and $final$.
The initial and final states of the TOSCA models are represented in CafeOBJ as follows:

```
module! STATEfuns {
  protecting(STATE)
  ...
  -- Many operator definitions explained in Section 7.1.1
  ..
  var SetND : SetOfNode
  var SetCP : SetOfCapability
  var SetRQ : SetOfRequirement
  var SetRL : SetOfRelationship
  var MP : PoolOfMsg
  var S : State

  pred init : State
  eq init(< SetND,SetCP,SetRQ,SetRL,MP >)
     = not (SetND = empND) and (MP = empMsg) and
       wfs(< SetND,SetCP,SetRQ,SetRL,MP >) and
       noNDCycle(< SetND,SetCP,SetRQ,SetRL,MP >) and
       allNDInStates(SetND,initial) and
       allCPInStates(SetCP,closed) and
       allRQInStates(SetRQ,unbound) .

  pred final : State
  eq final(< SetND,SetCP,SetRQ,SetRL,MP >)
     = allNDInStates(SetND,started) .

  pred wfs : State
  eq wfs(S)
     = wfs-uniqND(S) and wfs-uniqCP(S) and
       wfs-uniqRQ(S) and wfs-uniqRL(S) and
       wfs-allCPHaveND(S) and wfs-allRQHaveND(S) and
       wfs-allCPHaveRL(S) and wfs-allRQHaveRL(S) and
       wfs-allRLHaveCP(S) and wfs-allRLHaveRQ(S) and
       wfs-allRLHaveSameTypeCPRQ(S) and
       wfs-allRLNotInSameND(S) and
       wfs-allRLHoldLocality(S) and
       wfs-allNDHaveAtMostOneHost(S) .

  pred wfs-uniqND : State
  eq wfs-uniqND(< SetND,SetCP,SetRQ,SetRL,MP >)
     = uniqND(SetND) .
...
-- Similar fourteen definitions of wfs-*.
...
}
```

As described in Section 5.3.3, we need to define operators `DDSC` and `getAllObjInState` in order to use the Cyclic Dependency Lemma in the verification. Section 5.3.3 also describes two techniques to prove the invariant property of $noCycle(X, S)$. One is to design each transition rule to decrease dependencies between objects when it is applied. Section 6.6 shows example proofs using this technique.

Another technique used in this chapter is to design the system having a simpler constraint where some relationship between objects have no cyclic chains. Recalling Lemma 9, we can define DDSC to implement some simpler relationship r instead of the true DDS_C and use noCycle defined by using r instead of the true $noCycle_c$. Module STATECyclefuns defines an example of such DDSC:

```
module! STATECyclefuns {
  protecting(UtilFuns)

  var ND : Node
  var SetND : SetOfNode
  var SetCP : SetOfCapability
  var SetRQ : SetOfRequirement
  var SetRL : SetOfRelationship
  var MP : PoolOfMsg

  op getAllNDInState : State -> SetOfNode
  eq getAllNDInState(< SetND,SetCP,SetRQ,SetRL,MP >) = SetND .

  op DDSC : Node State -> SetOfNode
  eq DDSC(ND,< SetND,SetCP,SetRQ,SetRL,MP >)
  eq DDSC(ND,< SetND,SetCP,SetRQ,SetRL,MP >)
    = getNDsOfCPs(SetND,
                  getCPsOfRLs(SetCP,
                              getRLsOfRQs(SetRL,
                                          getRQsOfND(SetRQ,ND)))) .
}
```

Since this DDSC firstly finds the corresponding requirements of the given node, then finds the corresponding capabilities of the requirements, and finally finds and returns the parents of the capabilities, the true DDS_C is obviously a subset of this DDSC. Moreover this DDSC does not refer local states of objects and twelve transition rules of the example behavior model never change links of objects. It means that DDSC(X,S) for any reachable global state S from an initial state S_0 is the same as DDS(X,S_0) and noCycle defined by using DDSC is an invariant. noCycle can be defined using template module CYCLEPRED as follows:

```
module! STATEfuns {
  protecting(STATE)
  ...
  -- Other definitions explained above.
  ...
  extending(CYCLEPRED(
     STATECyclefuns {sort Object -> Node,
                     sort SetOfObject -> SetOfNode,
                     op empObj -> empND,
                     op getAllObjInState -> getAllNDInState})
     * {op noCycle -> noNDCycle}
     )
}
```

Step 0-2: Define *cont*. **Step 0-3:** Define *m*.

```
module! ProofBase {
  protecting(STATERules)
  vars S SS : State
  eq cont(S) = (S =(*,1)=>+ SS) .
```

Step 0-3: Define *m*.

```
  var SetND : SetOfNode
  var SetCP : SetOfCapability
  var SetRQ : SetOfRequirement
  var SetRL : SetOfRelationship
  var MP : PoolOfMsg
  op m : State -> Nat
  eq m(< SetND,SetCP,SetRQ,SetRL,MP >)
     = (#NodeInStates(initial,SetND) * 2)
     + (#NodeInStates(created,SetND) * 1)
     + (#NodeInStates(started,SetND) * 0)
     + (#CapabilityInStates(closed,   SetCP) * 2)
     + (#CapabilityInStates(open,     SetCP) * 1)
     + (#CapabilityInStates(available,SetCP) * 0)
     + (#RequirementInStates(unbound,SetRQ) * 2)
     + (#RequirementInStates(waiting,SetRQ) * 1)
     + (#RequirementInStates(ready,  SetRQ) * 0) .
```

Step 0-4: Define *inv*.

```
var SetND : SetOfNode
var SetCP : SetOfCapability
var SetRQ : SetOfRequirement
var SetRL : SetOfRelationship
var MP : PoolOfMsg
var S : State

pred inv-ifNDInitialThenRQUnboundReady : State
eq inv-ifNDInitialThenRQUnboundReady(< SetND,SetCP,SetRQ,SetRL,MP >)
   = ifNDInStatesThenRQInStates(SetND,initial,SetRQ,(unbound ready)) .
...
-- Many similar definitions of invariants.
-- 3 invariants are defined using predefined predicates.
-- 9 invariants are defined using problem specific predicates.
...

pred inv : State

-- wfs-*:
ceq inv(S) = false if not wfs-uniqND(S) .
...
-- Similar fourteen definitions for wfs-*.
...

-- inv-*:
ceq inv(S) = false if not inv-ifNDInitialThenRQUnboundReady(S) .
```

```
...
-- Similar eleven definitions for inv-*.
...
```

Step 0-5: Prepare for using the Cyclic Dependency Lemma.
For the CloudFormation example, the Cyclic Dependency Lemma is required to use for only one transition rule, `R01`. For the TOSCA example, however, there are two transition rules, `R01` and `R02` which cause cyclic situations in the verification. Thus, we need to define two lemmas in advance. One of them means that there is a contradiction when DDS_C of the specified `initial` resource includes any `initial` resources. Another means that there is a contradiction when DDS_C of the specified `created` resource includes any `created` resources. They are defined as the following two conditional equations:

```
ceq [CycleR01 :nonexec]:
   true = false
   if someNDInStates(DDSC(node(T:NDType,I:NDID,initial),S:State),initial) .

ceq [CycleR02 :nonexec]:
   true = false
   if someNDInStates(DDSC(node(T:NDType,I:NDID,created),S:State),created) .
```

Step 0-6: Prepare arbitrary constants.

```
  ops idND idND' idND1 idND2 idND3 : -> NDIDLt
  ops idCP idCP' idCP1 idCP2 idCP3 : -> CPIDLt
  ops idRQ idRQ' idRQ1 idRQ2 idRQ3 : -> RQIDLt
  ops idRL idRL' idRL1 idRL2 idRL3 : -> RLIDLt
  ops sND sND' sND'' sND''' : -> SetOfNode
  ops sCP sCP' sCP'' sCP''' : -> SetOfCapability
  ops sRQ sRQ' sRQ'' sRQ''' : -> SetOfRequirement
  ops sRL sRL' sRL'' sRL''' : -> SetOfRelationship
  ops tnd tnd' tnd'' tnd''' : -> NDType
  ops trl trl' trl'' trl''' : -> RLType
  ops snd snd' snd'' : -> NDState
  ops scp scp' scp'' : -> CPState
  ops srq srq' srq'' : -> RQState
  op stND : -> SetOfNDState
  op stCP : -> SetOfCPState
  op stRQ : -> SetOfRQState
  ops mp mp' : -> PoolOfMsg
  op msg : -> Msg
}
```

7.3.2 Lemmas for Using Cyclic Dependency Lemma

As described in Section 6.7, it is wise to define lemmas for using the Cyclic Dependency Lemma and use them in the similar cases. For this TOSCA example, two similar lemmas are required. One lemma claims that if there is an `initial` node in a reachable global state then there exists a transition rule applicable to the global state. Here we refer to it as the *initial-cont* lemma. It

can be proved as follows:

Step 1-0: Define a predicate to be proved.
Module `ProofInitialCont` defines the predicate as `invcont`. Note that we can only consider the case where `inv(S)` holds because `S` is a reachable global state.

```
module! ProofInitialCont {
  protecting(ProofBase)

  vars B1 B2 : Bool

  pred (_when _) : Bool Bool { prec: 64 r-assoc }
  eq (B1 when B2)
     = B2 implies B1 .

  var S: State

  pred invcont : State
  eq invcont(S)
    = cont(S) = true
    when inv(S) .
}
```

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

```
select ProofInitialCont .
:goal {
  eq invcont(< (node(tnd, idND, initial) sND), sCP, sRQ, sRL, mp >)
     = true .
}
```

Step 1-2: Think which rule is applied to the global state in the current case.
The applicable rule may be `R01` because the global state includes an `initial` node.

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.
The global state already matches to LHS of `R01`.

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

```
:csp {
  eq allHostedOnRQOfNDInStates(sRQ,idND,ready) = true .
  eq sRQ = (req(hostedOn,idRQ,unbound,idND) sRQ') .
  eq sRQ = (req(hostedOn,idRQ,waiting,idND) sRQ') .
}
-- Case 1: When all of the hostedOn requirements are ready:
:apply (rd) -- 1
-- Case 2: When there is an unbound hostedOn requirement of the node:
... -- More consideration needed.
-- Case 3: When there is a waiting hostedOn requirement of the node:
:apply (rd) -- 3
```

Only Case 2 remains unproved and it then becomes the current case.

Step 1-2: Think which rule is applied to the global state in the current case.
The applicable rule may be R04 because the global state in Case 2 includes an unbound hostedOn requirement.

Step 1-5: When there is a dangling link, split the case into cases where the linked object does or does not exist.

```
-- Case 2: When there is an unbound hostedOn requirement of the node:
:csp {
  eq onlyOneRLOfRQ(sRL,idRQ) = false .
  eq sRL = (rel(hostedOn,idRL,idCP,idRQ) sRL') .
}
-- Case 2-1: When the relationship of requirement idRQ does not exist:
:apply (rd) -- 2-1
-- Case 2-2: When the relationship of requirement idRQ exists:
:csp {
  eq existCP(sCP,idCP) = false .
  eq sCP = (cap(hostedOn,idCP,scp,idND') sCP') .
}
-- Case 2-2-1: When the capability of the relationship does not exist:
:apply (rd) -- 2-2-1
-- Case 2-2-2: When the capability of the relationship exists:
:ctf {
  eq idND' = idND .
}
-- Case 2-2-2-1: When the node of capability idCP is
--                               the same of requiement idRQ:
:apply (rd) -- 2-2-2-1
-- Case 2-2-2-2: When the node of capability idCP is not
--                               the same of requiement idRQ:
... -- More consideration needed.
```

Only Case 2-2-2-2 remains unproved.

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.

```
-- Case 2-2-2-2: When the node of capability idCP is not
--                               the same of requiement idRQ:
:csp {
  eq scp = closed .
  eq scp = open .
  eq scp = available .
}
-- Case 2-2-2-2-1: When the capability of idCP is closed:
... -- More consideration needed.
-- Case 2-2-2-2-2: When the capability of idCP is open:
:apply (rd) -- 2-2-2-2-2
-- Case 2-2-2-2-2: When the capability of idCP is avialable:
```

```
:apply (rd) -- 2-2-2-2-3
```

Only Case 2-2-2-2-1 remains unproved.

Step 1-2: Think which rule is applied to the global state in the current case.
The applicable rule may be R03 because the global state in Case 2-2-2-2-1 includes an unbound hostedOn requirement.

Step 1-5: When there is a dangling link, split the case into cases where the linked object does or does not exist.

```
-- Case 2-2-2-2-1: When the capability of idCP is closed:
:csp {
  eq existND(sND,idND') = false .
  eq sND = (node(tnd',idND',snd') sND') .
}
-- Case 2-2-2-2-1-1: When the node of the capability of idCP does not exist:
:apply (rd) -- 2-2-2-2-1-1
-- Case 2-2-2-2-1-2: When the node of the capability of idCP exists:
... -- More consideration needed.
```

Only Case 2-2-2-2-1-2 remains unproved.

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

```
-- Case 2-2-2-2-1-1: When the node of the capability of idCP does not exist:
:csp {
  eq snd' = initial .
  eq snd' = created .
  eq snd' = started .
}
-- Case 2-2-2-2-1-2-1: When the node of idND' is initial:
... -- More consideration needed.
-- Case 2-2-2-2-1-2-2: When the node of idND' is created:
:apply (rd) -- 2-2-2-2-1-2-2
-- Case 2-2-2-2-1-2-3: When the node of idND' is started:
:apply (rd) -- 2-2-2-2-1-2-3
```

Only Case 2-2-2-2-1-2-1 remains unproved.

Step 1-6: When falling in a cyclic situation, use the Cyclic Dependency Lemma.

```
-- Case 2-2-2-2-1-2-1: When the node of idND' is initial:
:init [CycleR01] by {
  T:NDType <- tnd;
  I:NDID   <- idND;
  S:State  <- < (node(tnd,idND,initial) sND), sCP, sRQ, sRL, mp >;
}
:apply (rd) -- 2-2-2-2-1-2-1
```

Thus, all cases are successfully proved and we can assume that *cont*(*S*) holds for any reachable global state *S* which include an `initial` node.

Another similar lemma claims that if there is a `created` node in a global state then there exists a transition rule applicable to the global state. Here we refer to it as the *created-cont* lemma. It can be proved as follows:

Step 1-0: Define a predicate to be proved.
Module `ProofCreatedCont` imports predicate `invcont` from module `ProofInitialCont` and additionally introduces the initial-cont lemma proved just above because the proof of this lemma uses it. Note that the `when` clause is omitted from the initial-cont lemma because `inv(S)` holds for any reachable global state `S`.

```
module! ProofCreatedCont {
  protecting(ProofInitialCont)

  var T : NDType
  var I : NDID
  var SetND : SetOfNode
  var SetCP : SetOfCapability
  var SetRQ : SetOfRequirement
  var SetRL : SetOfRelationship
  var M : PoolOfMsg
  -- This proof uses the initial-cont lemma.
  eq cont(< (node(T, I, initial) SetND),
             SetCP, SetRQ, SetRL, M >) = true .
}
```

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

```
select ProofCreatedCont .
:goal {
  eq invcont(< (node(tnd, idND, created) sND), sCP, sRQ, sRL, mp >)
     = true .
}
```

Step 1-2: Think which rule is applied to the global state in the current case.
The applicable rule may be `R02` because the global state includes a `created` node.

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.
The global state already matches to LHS of `RO2`.

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

```
:csp {
  eq allRQOfNDInStates(sRQ,idND,ready) = true .
  eq sRQ = (req(trl,idRQ,unbound,idND) sRQ') .
  eq sRQ = (req(trl,idRQ,waiting,idND) sRQ') .
}
-- Case 1: When all of the requirements are ready:
```

```
:apply (rd) -- 1
-- Case 2: When there is an unbound requirement of node idND:
... -- More consideration needed.
-- Case 3: When there is a waiting requirement of node idND:
... -- More consideration needed.
```

Both Case 2 and 3 remain unproved. Let Case 2 be the current state.

Step 1-2: Think which rule is applied to the global state in the current case.
The applicable rule may be R04, R07, or R11 because the global state in Case 2 includes an unbound requirement and the applicable rule depends on its type.

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.

```
-- Case 2: When there is an unbound requirement of node idND:
:csp {
  eq trl = hostedOn .
  eq trl = dependsOn .
  eq trl = connectsTo .
}
-- Case 2-1: When the type of requirement idRQ is hostedOn:
:apply (rd) -- 2-1
-- Case 2-2: When the type of requirement idRQ is dependsOn:
... -- More consideration needed.
-- Case 2-3: When the type of requirement idRQ is connectsTo:
... -- More consideration needed.
```

Case 2-1 is not a reachable global state because a node never becomes created when one of its hostedOn requirement is unbound, which makes inv(S) reduce to false. Thus, Case 2-2 and 2-3 remains unproved. Let Case 2-2 be the current case.

Step 1-5: When there is a dangling link, split the case into cases where the linked object does or does not exist.

```
-- Case 2-2: When the type of requirement idRQ is dependsOn:
:csp {
  eq onlyOneRLOfRQ(sRL,idRQ) = false .
  eq sRL = (rel(dependsOn,idRL,idCP,idRQ) sRL') .
}
-- Case 2-2-1: When the relationship of requirement idRQ does not exist:
:apply (rd) -- 2-2-1
-- Case 2-2-2: When the relationship of requirement idRQ exists:
:csp {
  eq existCP(sCP,idCP) = false .
  eq sCP = (cap(dependsOn,idCP,scp,idND') sCP') .
}
-- Case 2-2-2-1: When the capability of the relationship does not exist:
:apply (rd) -- 2-2-2-1
-- Case 2-2-2-2: When the capability of the relationship exists:
```

```
:ctf {
  eq idND' = idND .
}
-- Case 2-2-2-2-1: When the node of capability idCP is
--                                the same of requiement idRQ:
:apply (rd) -- 2-2-2-2-1
-- Case 2-2-2-2-2: When the node of capability idCP is not
--                                the same of requiement idRQ:
... -- More consideration needed.
```

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.

```
-- Case 2-2-2-2-2: When the node of capability idCP is not
--                                the same of requiement idRQ:
:csp {
  eq scp = closed .
  eq scp = open .
  eq scp = available .
}
-- Case 2-2-2-2-2-1: When capability idCP is closed:
... -- More consideration needed.
-- Case 2-2-2-2-2-2: When capability idCP is open:
:apply (rd) -- 2-2-2-2-2-2
-- Case 2-2-2-2-2-3: When capability idCP is available:
:apply (rd) -- 2-2-2-2-2-3
```

Case 2-2-2-2-2-2 and 2-2-2-2-2-2 are proved because `R07` is applicable. Thus, only Case 2-2-2-2-2-1 remains unproved.

Step 1-2: Think which rule is applied to the global state in the current case.
The applicable rule may be `R05` because the global state in Case 2-2-2-2-2-1 includes an `closed dependsOn` capability.

Step 1-5: When there is a dangling link, split the case into cases where the linked object does or does not exist.

```
-- Case 2-2-2-2-2-1: When capability idCP is closed:
:csp {
  eq existND(sND,idND') = false .
  eq sND = (node(tnd',idND',snd') sND') .
}
-- Case 2-2-2-2-2-1-1: When the node of capability idCP does not exist:
:apply (rd) -- 2-2-2-2-2-1-1
-- Case 2-2-2-2-2-1-2: When the node of capability idCP exists:
... -- More consideration needed.
```

Case 2-2-2-2-2-1-2 remains unproved.

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

```
-- Case 2-2-2-2-2-1-2: When the node of capability idCP exists:
:csp {
  eq snd' = initial .
  eq snd' = created .
  eq snd' = started .
}
-- Case 2-2-2-2-2-1-2-1: When node idND' is initial:
:apply (rd) -- 2-2-2-2-2-1-2-1
-- Case 2-2-2-2-2-1-2-2: When node idND' is created:
:apply (rd) -- 2-2-2-2-2-1-2-2
-- Case 2-2-2-2-2-1-2-3: When node idND' is started:
:apply (rd) -- 2-2-2-2-2-1-2-3
```

Note that Case 2-2-2-2-2-1-2-1 is proved by the initial-cont lemma, Case 2-2-2-2-2-1-2-2 and 2-2-2-2-2-1-2-3 are proved because R05 is applicable. Thus, Case 2-2 is proved and Case 2-3 reminds unprove. Case 2-3 is split into totally 21 cases all of which are proved similarly as split cases of Case 2-2.

Similarly Case 3 is split into totally 33 cases two of which require to use the Cyclic Dependency Lemma for rule R02. One of them is the following Case 3-2-2-2-2-2-2-2:

```
-- Case 3: When there is a waiting requirement of node idND:
...
-- Case splitting proceeds similarly as Case 2.
...
-- Case 3-2-2-2-2-2-2-2: When the node of idND' is created:
-- The global state in this case is
-- < (node(tnd,idND,created) node(tnd',idND',created) sND'),
--   (cap(dependsOn,idCP,open,idND') sCP'),
--   (req(dependsOn,idRQ,waiting,idND) sRQ'),
--   (rel(dependsOn,idRL,idCP,idRQ) sRL'),
--   mp >
:init [CycleR02] by {
  T:NDType <- tnd;
  I:NDID   <- idND;
  S:State  <- < (node(tnd,idND,created) sND), sCP, sRQ, sRL, mp >;
}
:apply (rd) -- 3-2-2-2-2-2-2-2
...
```

In this case, node idND is created and directly depends on node idND' which is also created. The Cyclic Dependency Lemma claims this global state is not reachable and this case is proved.

Another case is very similar to one above; the relationship type is not dependsOn but connectsTo as follows:

```
-- Case 3-3-2-1-2-2-2-2-2: When the node of idND' is created:
-- The global state in this case is
-- < (node(tnd, idND, created) node(tnd', idND', created) sND'),
--   (cap(connectsTo, idCP, open, idND') sCP'),
--   (req(connectsTo, idRQ, waiting, idND) sRQ'),
--   (rel(connectsTo, idRL, idCP, idRQ) sRL'),
--   mp >
```

```
:init [CycleR02] by {
  T:NDType <- tnd;
  I:NDID   <- idND;
  S:State  <- < (node(tnd,idND,created) sND), sCP, sRQ, sRL, mp >;
}
:apply (rd) -- 3-3-2-1-2-2-2-2-2
...
```

All other cases are successfully proved and we can assume that *cont*(*S*) holds for any reachable global state *S* which include a `created` node.

7.3.3 Proof of Condition (1)

Step 1-0: Define a predicate to be proved.
The proof of condition (1) requires to use the initial-cont lemma:

```
module! ProofInitCont {
  protecting(ProofBase)

  var S : State
  var T : NDType
  var I : NDID
  var SetND : SetOfNode
  var SetCP : SetOfCapability
  var SetRQ : SetOfRequirement
  var SetRL : SetOfRelationship
  var M : PoolOfMsg

  -- Predicate to be proved.
  pred initcont : State .
  eq initcont(S) = init(S) implies cont(S) .

  -- initial-cont lemma:
  eq cont(< (node(T, I, initial) SetND),
             SetCP, SetRQ, SetRL, M >) = true .
}
```

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

```
select ProofInitCont .
:goal {eq initcont(< sND, sCP, sRQ, sRL, mp >) = true .}
```

Step 1-2: Think which rule is applied to the global state in the current case.
The first rule is `R01`.

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.
Since LHS of rule `R01` requires the global state to have at least one `initial` node, the case is split into four more cases, i.e. no node, at least one `initial`, `created`, or `started` node.

```
:csp {
  eq sND = empND .
```

```
  eq sND = (node(tnd,idND,snd) sND') .
}
-- Case 1: When there is no node:
:apply (rd) -- 1
-- Case 2: When there is a node:
-- The state of the node is initial, created, or started.
:csp {
  eq snd = initial .
  eq snd = created .
  eq snd = started .
}
:apply (rd) -- 2-1
:apply (rd) -- 2-2
:apply (rd) -- 2-3
```

Case 2-1 is proved by the initial-cont lemma. In other cases, $init(S)$ does not hold for the global state S. Thus, sufficient condition (1) is proved.

7.3.4 Proof of Condition (2)

Step 2-0: Define a predicate to be proved.
The proof of condition (2) requires to use both of the initial-cont lemma and the created-cont lemma:

```
module! ProofContCont {
  protecting(ProofBase)

  vars S SS : State
  var CC : Bool
  var T : NDType
  var I : NDID
  var SetND : SetOfNode
  var SetCP : SetOfCapability
  var SetRQ : SetOfRequirement
  var SetRL : SetOfRelationship
  var M : PoolOfMsg

  -- Predicate to be proved.
  pred ccont : State State
  pred contcont : State
  eq ccont(S,SS)
     = inv(S) and not final(S) implies cont(SS) or final(SS) .
  eq contcont(S)
     = not (S =(*,1)=>+ SS if CC suchThat
            not ((CC implies ccont(S,SS)) == true)
         { true }) .

  -- initial-cont lemma:
  eq cont(< (node(T, I, initial) SetND),
     SetCP, SetRQ, SetRL, M >)
     = true .
```

```
    -- created-cont lemma:
    eq cont(< (node(T, I, created) SetND),
       SetCP, SetRQ, SetRL, M >)
       = true .
}
```

Step 2-1: Begin with the cases each of which matches to LHS of each rule.
The followings are cases for twelve transition rules:

```
  select ProofContCont .
  -- Goal of Condition (2) for rule R01
  :goal {
    eq contcont(< (node(tnd,idND,initial) sND), sCP, sRQ, sRL, mp >)
       = true .
  }

  -- Goal of Condition (2) for rule R02
  :goal {
    eq contcont(< (node(tnd,idND,created) sND), sCP, sRQ, sRL, mp >)
       = true .
  }

  -- Goal of Condition (2) for rule R03
  :goal {
    eq contcont(< sND, (cap(hostedOn,idCP,closed,idND) sCP), sRQ, sRL, mp >)
       = true .
  }

  -- Goal of Condition (2) for rule R04
  :goal {
    eq contcont(< sND,
                  (cap(hostedOn,idCP,available,idND) sCP),
                  (req(hostedOn,idRQ,unbound,idND')  sRQ),
                  (rel(hostedOn,idRL,idCP,idRQ)      sRL), mp >)
       = true .
  }

  -- Goal of Condition (2) for rule R05
  :goal {
    eq contcont(< sND, (cap(dependsOn,idCP,closed,idND) sCP), sRQ, sRL, mp >)
       = true .
  }

  -- Goal of Condition (2) for rule R06
  :goal {
    eq contcont(< sND, (cap(dependsOn,idCP,open,idND) sCP), sRQ, sRL, mp >)
       = true .
  }

  -- Goal of Condition (2) for rule R07
  :goal {
```

```
  eq contcont(< sND,
                (cap(dependsOn,idCP,scp,idND)      sCP),
                (req(dependsOn,idRQ,unbound,idND') sRQ),
                (rel(dependsOn,idRL,idCP,idRQ)     sRL), mp >)
     = true .
}

-- Goal of Condition (2) for rule R08
:goal {
  eq contcont(< sND,
                (cap(dependsOn,idCP,available,idND) sCP),
                (req(dependsOn,idRQ,waiting,idND')  sRQ),
                (rel(dependsOn,idRL,idCP,idRQ)      sRL), mp >)
     = true .
}

-- Goal of Condition (2) for rule R09
:goal {
  eq contcont(< sND, (cap(connectsTo,idCP,closed,idND) sCP), sRQ, sRL, mp >)
     = true .
}

-- Goal of Condition (2) for rule R10
:goal {
  eq contcont(< sND, (cap(connectsTo,idCP,open,idND) sCP), sRQ, sRL, mp >)
     = true .
}

-- Goal of Condition (2) for rule R11
:goal {
  eq contcont(< sND, sCP,
                (req(connectsTo,idRQ,unbound,idND) sRQ),
                (rel(connectsTo,idRL,idCP,idRQ)    sRL),
                (opMsg(idCP) mp) >)
     = true .
}

-- Goal of Condition (2) for rule R12
:goal {
  eq contcont(< sND, sCP,
                (req(connectsTo,idRQ,waiting,idND) sRQ),
                (rel(connectsTo,idRL,idCP,idRQ)    sRL),
                (avMsg(idCP) mp) >)
     = true .
}
```

The rest of this section describes the proof of condition (2) for rule R06 as an example.

```
select ProofContCont .
-- Goal of Condition (2) for rule R06
:goal {
```

```
  eq contcont(< sND, (cap(dependsOn,idCP,open,idND) sCP), sRQ, sRL, mp >)
     = true .
}
```

Step 2-7: When there is a dangling link, split the current case into cases where the linked object does or does not exist.

```
:csp {
  eq existND(sND,idND) = false .
  eq sND = (node(tnd,idND,snd) sND') .
}
-- Case 1: The node of capability idCP does not exist:
:apply (rd) -- 1
-- Case 2: The node of capability idCP exists:
```

Step 2-2: Split the current case for a rule into cases where the condition of the rule does or does not hold.

```
:csp {
  eq snd = initial .
  eq snd = created .
  eq snd = started .
}
-- Case 2-1: The node is initial:
:apply (rd) -- 2-1
-- Case 2-2: The node is created:
:apply (rd) -- 2-2
-- Case 2-3: The node is started:
```

Note that Case 2-1 and 2-2 are proved by the initial-cont lemma and the created-cont lemma respectively.

Step 2-3: Split the current case into cases where predicate $final$ does or does not hold in the next state.
We know that $final$ never holds in the next state of this case.

Step 2-4: Think which rule can be applied to the next state.
Since the next state in Case 2-3 includes an `available dependsOn` capability with identifier `idCP`, rule `R08` can be applied to it.

Step 2-7: When there is a dangling link, split the current case into cases where the linked object does or does not exist.

```
:csp {
  eq onlyOneRLOfCP(sRL,idCP) = false .
  eq sRL = (rel(trl,idRL,idCP,idRQ) sRL') .
}
-- Case 2-3-1: There is not a corresponding relationship:
:apply (rd) -- 2-3-1
-- Case 2-3-2: There is a corresponding relationship:
```

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.
LHS of rule R08 requires the type of the corresponding relationship to be dependsOn.

```
:csp {
  eq trl = hostedOn .
  eq trl = dependsOn .
  eq trl = connectsTo .
}
-- Case 2-3-2-1: The relationship is hostedOn:
:apply (rd) -- 2-3-2-1
-- Case 2-3-2-2: The relationship is dependsOn:
... -- More consideration needed.
-- Case 2-3-2-3: The relationship is connectsTo:
:apply (rd) -- 2-3-2-3
```

Only Case 2-3-2-2 remains unproved.

Step 2-7: When there is a dangling link, split the current case into cases where the linked object does or does not exist.

```
-- Case 2-3-2-2: The relationship is dependsOn:
:csp {
  eq existRQ(sRQ,idRQ) = false .
  eq sRQ = (req(trl',idRQ,srq,idND') sRQ') .
}
-- Case 2-3-2-2-1: There is not a corresponding requirement:
:apply (rd) -- 2-3-2-2-1
-- Case 2-3-2-2-2: There is a corresponding requirement:
... -- More consideration needed.
```

Only Case 2-3-2-2-2 remains unproved.

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.
LHS of rule R08 requires the type of the corresponding requirement to be dependsOn and the local state of it to be waiting.

```
-- Case 2-3-2-2-2: There is a corresponding requirement:
:csp {
  eq trl' = hostedOn .
  eq trl' = dependsOn .
  eq trl' = connectsTo .
}
-- Case 2-3-2-2-2-1: The requirement is hostedOn:
:apply (rd) -- 2-3-2-2-2-1
-- Case 2-3-2-2-2-2: The requirement is dependsOn:
:csp {
  eq srq = unbound .
  eq srq = waiting .
  eq srq = ready .
}
```

```
-- Case 2-3-2-2-2-2-2-1: The requirement is unbound:
... -- More consideration needed.
-- Case 2-3-2-2-2-2-2-2: The requirement is waiting:
:apply (rd) -- 2-3-2-2-2-2-2-2
-- Case 2-3-2-2-2-2-2-3: The requirement is ready:
:apply (rd) -- 2-3-2-2-2-2-2-3
-- Case 2-3-2-2-2-2-3: The requirement is connectsTo:
:apply (rd) -- 2-3-2-2-2-2-3
```

Only Case 2-3-2-2-2-2-2-1 remains unproved.

Step 2-4: Think which rule can be applied to the next state.
Since the next state in Case 2-3-2-2-2-2-2-1 includes an `unbound dependsOn` requirement with identifier `idRQ`, rule `R07` can be applied to it.

Step 2-7: When there is a dangling link, split the current case into cases where the linked object does or does not exist.

```
-- Case 2-3-2-2-2-2-2-1: The requirement is unbound:
:csp {
  eq existND(sND',idND') = false .
  eq sND' = (node(tnd',idND',snd') sND'') .
}
-- Case 2-3-2-2-2-2-2-1-1: The node of requirement idRQ does not exist:
:apply (rd) -- 2-3-2-2-2-2-2-1-1
-- Case 2-3-2-2-2-2-2-1-2: The node of requirement idRQ exists:
... -- More consideration needed.
```

Only Case 2-3-2-2-2-2-2-1-2 remains unproved.

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.
The global state already matches to LHS of `R07`.

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.

```
-- Case 2-3-2-2-2-2-2-1-2: The node of requirement idRQ exists:
:csp {
  eq snd' = initial .
  eq snd' = created .
  eq snd' = started .
}
-- Case 2-3-2-2-2-2-2-1-2-1: The node is initial:
:apply (rd) -- 2-3-2-2-2-2-2-1-2-1
-- Case 2-3-2-2-2-2-2-1-2-2: The node is created:
:apply (rd) -- 2-3-2-2-2-2-2-1-2-2
-- Case 2-3-2-2-2-2-2-1-2-3: The node is started:
:apply (rd) -- 2-3-2-2-2-2-2-1-2-3
```

All cases are successfully proved.

7.3.5 Proof of Condition (3)

Step 3-0: Use natural number axioms.
The framework provides a module, NATAXIOM, which defines several natural number axioms to be used for proof of condition(3) and (4). Module ProofMeasure should protecting import NATAXIOM as well as ProofBase:

```
module! ProofMeasure {
  protecting(ProofBase)
  protecting(NATAXIOM)
```

Step 3-1: Define a predicate to be proved.

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

  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.
Here we show the proof of condition (3) for rule R06 as an example:

```
-- Goal of Condition (3) for rule R06
select ProofMeasure .
:goal {
  eq mesmes(< sND, (cap(dependsOn,idCP,open,idND) sCP), sRQ, sRL, mp >)
     = true .
}
```

Step 3-3: Split the current case for a rule into cases where the condition of the rule does or does not hold.

```
:ctf {
  eq state(getNode(sND,idND)) = started .
}
:apply (rd) -- 1
:apply (rd) -- 2
```

Condition (3) for other rules can be similarly proved.

7.3.6 Proof of Condition (4)

Step 4-0: Use natural number axioms.
Module ProofMesFinal should protecting import NATAXIOM as well as ProofBase:

```
module! ProofMesFinal {
  protecting(ProofBase)
  protecting(NATAXIOM)
```

Step 4-1: Define a predicate to be proved.

```
var S : State
pred mesfinal : State .
eq mesfinal(S)
   = inv(S) and cont(S) and m(S) = 0 implies final(S) .
```

Step 4-2: Begin with the cases each of which matches to LHS of each rule.
Here we show the proof of condition (3) for rule `R06` as an example:

```
-- Goal of Condition (3) for rule R06
select ProofMesFinal .
:goal {
  eq mesfinal(< sND, (cap(dependsOn,idCP,open,idND) sCP), sRQ, sRL, mp >)
     = true .
}
```

Step 4-3: Split the current case for a rule into cases where the condition of the rule does or does not hold.

```
:ctf {
  eq state(getNode(sND,idND)) = started .
}
:apply (rd) -- 1
:apply (rd) -- 2
```

Condition (4) for other rules can be similarly proved.

7.4 Evaluation

Rate of Reuse

We need 37 sorts to represent the TOSCA structure models and all of them can be just instantiated and renamed from predefined sorts provided by the framework.

We also need totally 218 predicates/operators not including definitions of arbitrary constants used in proofs. 104 of them can be just instantiated and renamed form predefined operators. 11 predicates, such as `initcont`, `contcont`, and so on, are the same as in proofs of the CloudFormation example and so can be copied from them. 27 state predicates are simple wrappers of other predicates, such as `wfs-*` and `inv-*`.

Thereby, 76 operators are problem specific ones. 23 of them are constructors including local state and type literals. 15 operators are selectors such as `id`, `type`, `state`, and ones for links. The framework requires users to define 8 operators, i.e. `init`, `final`, `wfs`, `inv`, `m`, `invK`, `getAllNDInState`, and `DDSC`.

Remaining 30 operators are fully original ones of this problem, however almost all of them can be easily defined combining predefined operators and can be written in only several code lines. As described in Section 7.1.1, there are three kinds of them:

- Check the consistency between messages and local states of objects, e.g. if there is an available message then the corresponding capability should be available. Currently, the framework provides no functionality to support messaging mechanisms.
- Check the consistency between capabilities and requirements connected by relationships. The framework provides many operators and lemmas for links but does not provide those for chains of links.
- Check other problem-specific constraints, e.g. every node should be hosted on exactly one VM node.

Size of Codes

The representation of the TOSCA model in CafeOBJ consists of about 600 lines of codes not including comment lines. About 530 lines of codes represent the structure model and 70 lines represent twelve rules. We estimate that the size of the structure model representation is 40% compared to when we would code it without using the framework, whereas the size of the behavior model (transition rules) is the same.

The size of codes for proofs is essentially the same as when not using the framework because reusable codes for proofs are proved lemmas provided by the framework. The proofs of the TOSCA example need 36 lemmas, 13 of which are already proved by the framework in a general level of abstraction. Reminding 23 lemmas are required to prove condition (5) and (6) for invariants about three kinds of problem specific operators described above.

Consistent Structure of Proof

However the framework does not remarkably reduce the size of codes, time and efforts to develop them is radically reduced. Of course, it is mainly because this is our second experience of the same problem, whereas the previous proof scores did not have any unified policies of splitting and so were very difficult to understand even for us. The framework makes the new proof scores become much clear, especially those of conditions (2)(3)(6) which should be proved for each of twelve trans rules.

The recommended module structure also help to make proof scores easier to understand. We can instantly find the place where something is defined and can instantly imagine which parts of the proof may be affected when something is modified.

Similarly as application frameworks of software development, our framework not only provides reusable entities to reduce the size of codes of proof but also guides users how to design the models and how to systematically think and develop proofs, which brings high productivity by minimizing development efforts and high maintainability by consistent structure of models and proofs.

Chapter 8

Related Work and Conclusion

8.1 Related Work

Formal Approach for Cloud Orchestration

Salaün, G., et al. [6, 19, 20] designed a system setup protocol and demonstrated to verify a liveness property of the protocol using their model checking method. Although their setup protocol is essentially the same as the behavior model of our TOSCA example in this paper, there are two main differences. Firstly, their protocol is based on a specific implementation which challenges distributed management of cloud resources while current popular implementations, e.g. CloudFormation, use centralized management. On the other hand, our model is rather abstract without assuming distributed or centralized implementations. Secondly, they used model checking while we use theorem proving. They checked about 150 different models of system including from four to fifteen components in which from 1.4 thousand to 1.4 million transitions are generated and checked. They found a bug of their specification because checked models fortunately included error cases. The model checking method can verify correctness of checked models and so they should include all boundary cases. In our formalization, the specification itself is verified by interactive theorem proving in which all boundary cases are necessary in consideration in a systematic way. It achieves structural and deep understanding that is required to develop trusted systems.

Dependency Management between Internal Resources

CloudFormation and OpenStack Heat can manage resources on the IaaS layer, however, they support to manage dependencies between resources in VMs. For example, suppose a software component (SC_1) on a VM (VM_1) can be activated only after waiting for activation of another component (SC_2) on another VM (VM_2), CloudFormation requires a pair of special purpose resources, namely, *WaitCondition* and *WaitConditionHandle*. VM_1 should be declared to depend on the WaitCondition resource. The corresponding WaitConditionHandle resource provides a URL that should be passed to the script for initializing VM_2. When SC_2 is successfully activated, the script sends a success signal to the URL, which causes the WaitCondition become active and then creation of dependent VM_1 starts. This style of management includes several problems. Firstly, it forces complicated and troublesome coding of operations. Secondly, although only SC_1 should wait for SC_2, all other components on VM_1 are also forced to wait. This causes unnecessary slowdown of system creation. Thirdly, it tends to make cyclic dependencies. Suppose SC_2 should also wait for another component SC_3 on VM_1. Although the

dependency among components, SC_1, SC_2, and SC_3 is acyclic, the dependency between VMs is cyclic. This may be solved by splitting VM_1 to two VMs, one is for SC_1 and another is for SC_3, but it causes increased cost and delayed creation. Our formalization can manage any types of resources and solve this kind of problems in a smarter way because it can manage finer grained dependencies, which is shown as invocation rules described in Section 7.2.

Next Version of OASIS TOSCA

OASIS TOSCA TC currently discusses the next version (v1.1) to define a standard set of nodes, relationships, and operations [15]. It is planned to use state machines to describe behavior of the standard operations, which is a similar approach as ours. However, the usage is limited to clarify the descriptions of the standard and the way for type architects to define behavior of their own types is out of the scope of standardization. We provide a way to specify behavior of types and show that it can be used for verification.

8.2 Future Issues

While more than six seventh of operators and one third of lemmas for the TOSCA example can be easily defined using predefined operators and proved lemmas, several extensions of our framework are desired to further reduce problem specific coding and proving. The general formalization for messaging mechanism and chains of links is required.

CloudFormation provides a default roll back mechanism when an operation failure occurs but it requires manual operations when the roll back also fails. On the other hand, the current version of TOSCA does not manage operation failures and it focuses on declaratively defining expected configurations of cloud systems. A possible future extension of TOSCA may be to define alternative configurations in failure cases, which we think we can easily extend our formalization to handle.

In this paper, we explain our framework using examples of system setup operations of cloud systems because cloud orchestration tools currently focus on them. However, TOSCA is designed to be used for any types of system operations such as scale-out and scale-in. One of the main difficulties to specify scaling operations is that they dynamically change the structure of cloud systems, for which our framework should be enforced from two points of view. Firstly, some additional guidance is required to design state measuring functions, especially for the case of scale-out where the number of resources in the system will increase. Secondly, while the user of our framework is left responsible for proving the invariant property of *noCycle*, it may be not a trivial work as to dynamic structure. Some constraint should be introduced in the cloud system structure to keep acyclicness of dependency. One possible solution is to assume a partial order of types of objects and to allow transition rules to produce dependency only in the descending order. Two techniques to prove the invariant property of *noCycle* described in Section 5.3.3 will be also effective for the solution.

8.3 Conclusion

A general formalization of declarative cloud orchestration is proposed and a framework is provided for interactive developing proof scores. The framework provides (1) a general way to formalize specifications of different kinds of cloud orchestration tools and (2) a procedure for

how to verifying leads-to properties, as well as invariant properties, of formalized specifications. It also provides (3) general templates and libraries of formal descriptions for specifying orchestration of cloud systems and (4) proved lemmas for general predicates of the libraries to be used for verification.

The framework has been applied to the verification of specifications of AWS CloudFormation and also of OASIS TOSCA. The provided procedure systematically assists the verification process and makes its generic part be routine work whose efforts are reduced by the provided logic templates and predicate libraries. As a result, a verification engineer can concentrate on the work specific to the individual problem, which brings high productivity by minimizing development efforts and high maintainability by consistent structure of models and proofs.

A related work applied their model checking method to a typical problem in the domain of cloud orchestration, in which many of finite-state systems were checked. Our framework is more general to be applied to different kinds of models in the domain and to be used for interactive theorem proving which can verify systems of arbitrary many number of states in a significantly systematic way.

An example of usage of our formalization shows a general way to manage dependencies of cloud resources which is a smarter one than that of the most popular tool, AWS CloudFormation.

It is also demonstrated that the framework can be used to specify, represent, and verify the behavior models of the standard specification language, OASIS TOSCA, of cloud orchestration where the standard has not yet provided any ways to do so.

The major contributions of this paper are (1) it shows that cloud orchestration is a practical and suitable domain to apply interactive theorem proving and (2) it introduces the idea of frameworks from software development to proof development which results in high productivity and high maintainability of proofs.

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

Bibliography

[1] Amazon Web Services. AWS CloudFormation - Infrastructure as Code & AWS Resource Provisioning. http://aws.amazon.com/cloudformation/, Accessed: 2016-06-10.

[2] CafeOBJ. CafeOBJ Algebraic Specification and Verification. https://cafeobj.org/, Accessed: 2016-06-10.

[3] K. Mani Chandy and Jayadev Misra. *Parallel program design - a foundation*. Addison-Wesley, 1989.

[4] Chef Software Inc. Chef |IT Automation for speed and awesomeness. https://www.chef.io/chef/, Accessed: 2016-06-10.

[5] Clark Evans, Brian Ingerson, Oren Ben-Kiki. YAML Ain ' t Markup Language (YAML) Version 1.2. http://www.yaml.org/spec/1.2/spec.html, Accessed: 2016-11-05.

[6] Xavier Etchevers, Thierry Coupaye, Fabienne Boyer, and Noel De Palma. Self-configuration of distributed applications in the cloud. In *2013 IEEE Sixth International Conference on Cloud Computing*, pages 668–675, 2011.

[7] Kokichi Futatsugi. Generate & check method for verifying transition systems in cafeobj. In *Software, Services, and Systems*, LNCS 8950, pages 171–192. Springer, 2015.

[8] Kokichi Futatsugi, Daniel Găină, and Kazuhiro Ogata. Principles of proof scores in CafeOBJ. In *Theoretical Computer Science*, volume 464, pages 90–112. Elsevier, 2012.

[9] Joseph Goguen. OBJ Family/ OBJ3 CafeOBJ Maude Kumo FOOPS Eqlog. http://cseweb.ucsd.edu/~goguen/sys/obj.html, Accessed: 2016-06-10.

[10] David Heinemeier Hansson. Ruby On Rails. http://rubyonrails.org/, Accessed: 2016-06-09.

[11] Internet Engineering Task Force (IETF). The JavaScript Object Notation (JSON) Data Interchange Format. https://tools.ietf.org/html/rfc7159, Accessed: 2016-03-26.

[12] Yukihiro Matsumoto. Ruby Programming Language. https://www.ruby-lang.org/, Accessed: 2016-11-05.

[13] Maude. The maude system. http://maude.cs.uiuc.edu/, Accessed: 2016-06-10.

[14] OASIS. TOSCA - Topology and Orchestration Specification for Cloud Applications Version 1.0. http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.pdf, Accessed: 2016-06-10.

[15] OASIS. TOSCA Simple Profile in YAML Version 1.0. http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/cs01/TOSCA-Simple-Profile-YAML-v1.0-cs01.pdf, Accessed: 2016-06-15.

[16] Object Management Group. Business Process Model & Notation - Resource Page. http://www.omg.org/bpmn/index.htm, Accessed: 2016-11-13.

[17] Puppet. Puppet - The shortest path to better software. https://puppet.com/, Accessed: 2016-06-10.

[18] Red Hat Inc. Ansible is Simple IT Automation. http://www.ansible.com/, Accessed: 2016-06-10.

[19] Gwen Salaün, Fabienne Boyer, Thierry Coupaye, Noel De Palma, Xavier Etchevers, and Olivier Gruber. An experience report on the verification of autonomic protocols in the cloud. *Innovations in Systems and Software Engineering*, 9(2):105–117, 2013.

[20] Gwen Salaün, Xavier Etchevers, Noel De Palma, Fabienne Boyer, and Thierry Coupaye. Verification of a self-configuration protocol for distributed applications in the cloud. In *Assurances for Self-Adaptive Systems*, LNCS 7740, pages 60–79. Springer, 2013.

[21] The OpenStack project. Heat - OpenStack. https://wiki.openstack.org/wiki/Heat, Accessed: 2016-06-10.

[22] Guido van Rossum. Python Software Foundation. https://www.python.org/, Accessed: 2016-11-05.

Publications

[1] Hiroyuki YOSHIDA, Kazuhiro OGATA, and Kokichi FUTATSUGI, Formalization and Verification of Declarative Cloud Orchestration, *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Proceedings*, Lecture Notes in Computer Science 9407, pp 33-49, Springer, Paris, France, November 3-5, 2015