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: Research 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. 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 a kind of liveness properties of formalized specifications. It also provides (a) general templates and libraries of formal descriptions for specifying orchestration of cloud systems and (b) logical proofs of lemmas for general predicates of the libraries.

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.

Acknowledgments

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.

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. However cloud computing enables to easily, cheaply, and repeatedly prepare testing environments for applications, automated operations intrinsically cannot be tested on testing environments; they should be tested only on production environments.

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* [16]. CloudFormation can manage resources provided by IaaS platform of AWS, such as VMs (EC2), block storages (EBS), load balancers (ELB), and so on. CloudFormation automatically sets up these resources according to a *template* that declaratively defines dependencies of resources. 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* [12], *Chef* [3], and *Ansible* [13]. 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* [11] that can be used to describe the structure of both types of resources, on IaaS and inside VMs. The resource structure is called a *topology* and a TOSCA tool is expected to automate system operations based on resource dependencies declaratively defined in topologies. Currently, however, there

is no practical implementation of declarative specifications of TOSCA because it has not yet explicitly provided any way to specify behavior of a topology, i.e. how to automate a topology.

We believe formal approaches will provide systematic ways to guarantee the 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 size 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 to think 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) [8] 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, i.e. 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 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) and a compatible open source tool is being developed as *OpenStack Heat* [16]. 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. 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.

CloudFormation models a cloud system simply as a set of *resources* on IaaS platform of AWS. The model is called a *template*. A resource has an identifier and a type and includes several *properties* which may depend on other resources. CloudFormation automates to setup a cloud system according to the specified dependency of the resources. Fig. 2.1 is a part of a very simple CloudFormation template written in JSON format [9]. 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.

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

Figure 2.1: A Very Simple CloudFormation Template

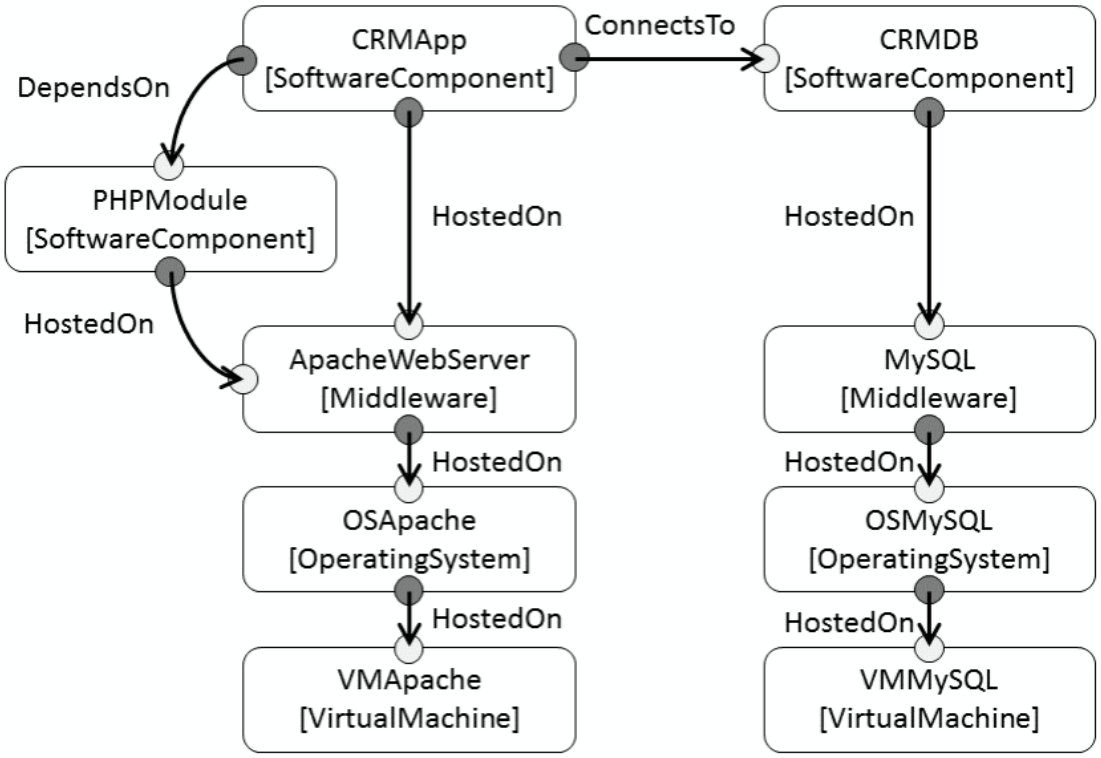

Figure 2.2: An Example of TOSCA topology

2.2 OpenStack Heat

2.3 Puppet, Chef, Ansible

2.4 OASIS TOSCA

TOSCA is a language to define a *service template* for a cloud system[1]. A service template consists of a *topology template* and optionally a set of *plans*. A topology template defines the resource structure of a cloud application. Note that a topology template can be parameterized to give actual environment parameters such as IP addresses. It 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 application, such as a setup plan, written by a standard process modeling language, such as BPMN.

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. 2.2 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. Fig 2.3 is a part of the topol-

[1]TOSCA says a "service" to mean a cloud system.

```
<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 2.3: A Topology Template of TOSCA

ogy template of Fig. 2.2. In this example, there are two nodes (`VMApache` and `OSApache`) and one relationship. 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. In this example, `VMApacheOS` is a capability of `VMApache` and `OSApacheContainer` is a requirement of `OSApache`. A relationship relates a requirement of a source node to a capability of a target node. 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.

In a typical scenario, a type architect defines and provides several types of those elements and an application architect uses them to define a topology of a cloud application. The type architect also defines operations 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 application 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 depends on 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.

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 [7] language family, such as Maude [10]. 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
  op _\in_ : Elt.X Set -> Bool
  op _A_ : 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.

```
  op _\\_ : Set 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) .
}
```

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 it is not allowed to add and collapse elements of the imported modules. `extending` means it is allowed only to add but not to collapse them. `using` means it is allowed to add and collapse them.

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 `_\_` are defined which mean union($\cup$), inclusion($\in$), intersection($\cap$), and difference($\setminus$) of sets respectively.

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`. 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. Thus, `#(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. A 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 states. Transition rule c2o specifies that a closed service appearing in a state is changed to open, and o2r specifies that an open service is changed to ready if there is at least one other service; ctrans means "conditional trans". Command execute makes CafeOBJ try to apply transition rules until no one can be applied.

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

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

Rule c2o makes 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 state < sv(1,closed) > because it has only one element.

3.3 Search Predicates

What is called search predicates can be used to conduct reachability analysis for such state machines specified in CafeOBJ . Let us consider the following code fragment:

```
open 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.
close
```

By reducing the term in the code fragment, CafeOBJ finds any next states of the given 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 state such that some conditions hold in those next states. Let us consider the following code fragment[4]:

```
open UPDATE .
 pred anyOpen : SvSet .
   -- The same as: op anyOpen : SvSet -> Bool .
 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.
```

Here, pred declares a predicate, i.e., an operator whose coarity is Bool. The reduction returns true in which CafeOBJ finds any next states of the given state such that an open service is appearing. In this case, transition rule c2o makes such next state. Note that when the predicate tries a conditional transition rule, it binds the rule's condition to CC. 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 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 state of the given state such that NO open service is appearing. The reduction proceeds as follows:

1. Try to match LHS of c2o to the given 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, try to match LHS of o2r to the given state, the condition (i.e. # SVS > 0) to CC, and the substituted RHS (i.e. < sv(2,ready) sv(1,closed) >) to < SVS >.

[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 states reachable from the given state with at most three state transitions. If * is used instead of +, CafeOBJ also includes the given 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.

5. Evaluate the substituted `suchThat` clause which reduces to true because `sv(2,ready) sv(1,closed)` does not include any open service.

6. Then the search predicate returns true and the whole term reduces to false.

This means that there is a next states of state `< sv(1,closed) sv(2,open) >` which does not include any open services; CafeOBJ finds that it is 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 .
                             -- to anyOpen(sv(1,closed)) = true .
reduce anyOpen(sv(1,closed)) == true .
                             -- 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 sentence 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.

3.4 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 [6]. 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 state *S* when at least two services included in *S* are not `ready`.

```
module! ProofUPDATE {
 protecting(UPDATE)

 -- Theorem to be proved.
 pred theorem : State

 vars N N1 N2 : Nat
 vars St1 St2 : LocalState .
 vars SVS SVS' : SvSet

 eq theorem(< sv(N1,St1) sv(N2,St2) SVS >)
   = ((St1 == ready) = false and (St2 == ready) = false)
     implies < sv(N1,St1) sv(N2,St2) SVS > =(*,1)=>+ < SVS' > .

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

```
 -- Arbitrary constants.
 op s : -> State
 ops sv1 sv2 : -> Service
 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 requires an axiom for natural numbers which says that the successor of a natural number is greater than 0.

Firstly, we begin with the most general case; the state is < sv1 sv2 svs > where sv1 and sv2 are arbitrary constants of sort Service and svs is of SvSet.

```
-- The most general case.
open ProofUPDATE .
 eq s = < sv1 sv2 svs > .
 reduce theorem(s) . -- to false.
close
```

This case is too general to judge whether the theorem does or does not hold. 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 s = < sv1 sv2 svs > .
 eq sv1 = sv(n1,closed) .
 eq sv2 = sv(n2,closed) .
 reduce theorem(s) . -- to true.
close

-- Case 2: Both services are open.
open ProofUPDATE .
 eq s = < sv1 sv2 svs > .
 eq sv1 = sv(n1,open) .
 eq sv2 = sv(n2,open) .
 reduce theorem(s) . -- to true.
close

-- Case 3: A closed service and an open service.
open ProofUPDATE .
 eq s = < sv1 sv2 svs > .
 eq sv1 = sv(n1,closed) .
 eq sv2 = sv(n2,open) .
 reduce theorem(s) . -- 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. Thus, 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 a 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 case 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 each of which `eq` LHS_i `=` RHS_i `.` is added to.

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

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

- `:apply (`$name_1$ $name_2$`)`
  Combine named case splittings. When $name_1$ splits n cases and $name_2$ splits m case, the current case is split into totally $n \times m$ cases. It can also specify `rd`, i.e. `:apply (`n_1 n_2 `rd)`, which means to reduce 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.

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

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

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

Command `select` is similar to `open` except that it does not allow to declare new sorts, operators, equations, and so on.

Firstly, the goal to be proved should represent the most general case. Note that predicate `theorem` is defined by only one equation, which implicitly means that it does not hold for global states which does not match the LHS of the rule, i.e. `< sv(N1,St1) sv(N2,St2) SVS >`. Thus, the state in the most general case is `< sv(n1,st1) sv(n2,st2) svs >` where `n1, st1, n2, st2`, and `svs` are arbitrary constants of corresponding classes. 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. Thus 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(< (sv(n1, st1) sv(n2, st2) svs) >)
          = true .
[csp-st1] 1*
    -- context module: #Goal-1
    -- assumption
      eq [csp-st1]: st1 = closed .
    -- targeted sentence:
      eq theorem(< (sv(n1, st1) sv(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(< (sv(n1, st1) sv(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(< (sv(n1, st1) sv(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(< (sv(n1, st1) sv(n2, st2) svs) >)
          = true .
[csp-st1] 2*
    -- context module: #Goal-2
    -- assumption
      eq [csp-st1]: st1 = open .
    -- targeted sentence:
      eq theorem(< (sv(n1, st1) sv(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(< (sv(n1, st1) sv(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(< (sv(n1, st1) sv(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(< (sv(n1, st1) sv(n2, st2) svs) >)
          = true .
[csp-st1] 3*
    -- context module: #Goal-3
    -- assumption
      eq [csp-st1]: st1 = ready .
```

```
    -- targeted sentence:
      eq theorem(< (sv(n1, st1) sv(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(< (sv(n1, st1) sv(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(< (sv(n1, st1) sv(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(< (sv(n1, st1) sv(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 to abstractly and formally specify the structure and behavior. A specified model is represented by a formal specification language, i.e. 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*, which is represented by XML as illustrated in Fig 2.3. 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 show 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 but its ID is hidden and we assume it is MyEIP::InsID since its parent is MyEIP and its type

[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.

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. Note that a link is represented by an identifier of the linked object in our framework.

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 a set of possible types, a 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. In this example, the initial states of resource and property objects are assumed as `initial` and `notready` respectively. 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.

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. The behavior is very simple; CloudFormation tries to start all resources according to the dependency specified by the template. This can be modeled such that a local state of a resource is firstly *initial* and finally *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*.

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 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]

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

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

```
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 >
 => < (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)` will be explained in Section 5.2, however, it checks a set of properties `SetPR` whether every property of resource `IDRS` is `ready`. Rule

R01 means that an initial resource becomes started when all of its properties are ready. 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 following is a 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 only 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,
```

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

```
      op getObject -> getResource,
      op allObjInStates -> allRSInStates,
      op allObjNotInStates -> allRSNotInStates,
      op someObjInStates -> someRSInStates}
  )

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

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

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

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

The following is a list of part of sorts and 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:

- sort `Object` (as `Resource`)
  Sort for objects themselves.

- sort `ObjIDLt` (renamed 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 _ == _ .

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

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

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

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

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

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

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

- op `empObj` (as `empRS`)
  Constant representing an empty set of objects.

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

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

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

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

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

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

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

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

- op `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 be extended 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 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].

Similary, 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
```

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

```
  -- Selectors
  op parent : Property -> RSID
  op refer : Property -> RSID
  eq type(prop(TPR,IDPR,SPR,IDRS1,IDRS2)) = TPR .
  eq id(prop(TPR,IDPR,SPR,IDRS1,IDRS2)) = IDPR .
  eq state(prop(TPR,IDPR,SPR,IDRS1,IDRS2)) = SPR .
  eq parent(prop(TPR,IDPR,SPR,IDRS1,IDRS2)) = IDRS1 .
  eq 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 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 a list of eleven sorts and 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:

- sort `Object` (actually named as `Property`)
  Sort for linking objects.
- sort `ObjID` (as `PRID`)
  Sort for identifiers of linking objects.
- sort `ObjType` (as `PRType`)
  Sort for types of linking objects.
- sort `ObjState` (as `PRState`)
  Sort for local states of linking objects.
- sort `SetOfObject` (as `SetOfProperty`)
  Sort for sets of linking objects.
- sort `SetOfObjState` (as `SetOfPRState`)
  Sort for sets of local states of linking objects.
- sort `LObject` (as `Resource`)
  Sort for linked objects.
- sort `LObjID` (as `RSID`)
  Sort for identifiers of linked objects.
- sort `LObjState` (as `RSState`)
  Sort for local states of linked objects.
- sort `SetOfLObject` (as `SetOfResource`)
  Sort for sets of linked objects.

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

- op `link` (as `parent` and `refer`)
  Selector used as `link`(*obj*) which returns the identifier of the object linked by *obj*.

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

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

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

- op `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 a list of part of operators predefined by template module `OBJLINKMANY2ONE` whereas argument *obj* is a linking object, *seto* is a set of linking objects, *setls* is a set of local states of linking objects, *lobj* is a linked object, *lid* is an identifier of a linked object, *setlo* is a set of linked objects, and *setlls* is a set of local states of linked objects:

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

- op `getXOfZ` (as `getRSOfPR` and `getRRSOfPR`)
  Operator used as `getXOfZ`(*setlo*,*obj*) which returns an object linked by *obj* included in *setlo*.

- op `getZsOfX` (as `getPRsOfRS` and `getPRsOfRRS`)
  Operator used as `getZsOfX`(*seto*,*lobj*) which returns a set of objects linking to *lobj* included in *seto*.

- op `getZsOfXInStates` (as `getPRsOfRSInStates` and `getPRsOfRRSInStates`)
  Operator used as `getZsOfXInStates`(*seto*,*lobj*,*setls*) which returns a set of objects linking to *lobj* included in *seto* and whose local states are included in *setls*.

- op `getXsOfZs` (as `getRSsOfPRs` and `getRRSsOfPRs`)
  Operator used as `getXsOfZs`(*setlo*,*seto*) which returns a set of objects linked by some object included in *seto* included in *setlo*.

- op `getXsOfZsInStates` (as `getRSsOfPRsInStates` and `getRRSsOfPRsInStates`)
  Operator used as `getXsOfZsInStates(`*setlo*`,`*seto*`,`*setlls*`)` which returns a set of objects linked by some object included in *seto* included in *setlo* and whose local states are included in *setlls*.

- op `getZsOfXs` (as `getPRsOfRSs` and `getPRsOfRRSs`)
  Operator used as `getZsOfXs(`*seto*`,`*setlo*`)` which returns a set of objects linking to some object included in *setlo* included in *seto*.

- op `getZsOfXsInStates` (as `getPRsOfRSsInStates` and `getPRsOfRRSsInStates`)
  Operator used as `getZsOfXsInStates(`*seto*`,`*setlo*`,`*setls*`)` which returns a set of objects linking to some object included in *setlo* included in *seto* whose local states are included in *setls*.

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

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

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

- op `ifXInStatesThenZInStates`
  (as `ifRSInStatesThenPRInStates` and `ifRRSInStatesThenPRInStates`)
  Predicate used as `ifXInStatesThenZInStates(`*setlo*`,`*setlls*`,`*seto*`,`*setls*`)` which checks whether 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.

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 expansive 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 a 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 of objects* `S'` *such that* `S=(O' S')` *holds and* `P(O')` *does not hold.*

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 of objects* `S'` *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.

For example, allZOfXInStates is instantiated from allObjP where P(O) checks whether 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 case splitting can be specified as follows:

```
: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* Sx *be a set of linking objects,* Sz *be a set of linked objects,* STx *be a set of local states of linking objects,* STz *be a set of local states of linked objects, and* St *be a local state of linking object where* St *is not included in* STx*. Then,* allObjInStates(Sx,St) *implies* ifXInStatesThenZInStates(Sx,STx,Sz,STz).

This lemma is represented in CafeOBJ as follows[3]:

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

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

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:

```
eq inv1(< SetRS,SetPR >) =
  ifRSInStatesThenPRInStates(SetRS,started,SetPR,ready) .
```

In order to show that `inv1` is an invariant, we need a lemma which says that if all resources are `initial` then `inv1` holds. The lemma is defined as follows:

```
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 got by renaming `m2o-lemma07` as follows:

```
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 omitted when renamed. Using the implication lemma, this lemma can be define as follows:

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

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

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

This lemma is represented in CafeOBJ as follows:

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

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 that `inv1` above is an invariant, we also need another lemma which says that `inv1` keeps to hold when rule `R02` is applied and makes a property transit from `notready` to `ready`. The lemma is defined as follows:

```
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 the antecedent 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 got by renaming `m2o-lemma11` as follows:

```
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:

```
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 trans rule and the system 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).

The rest of this section will describe a formal definition of cyclic dependency and show examples using the simple case shown in Fig. 2.1 and transition rules `R01` and `R02` in Section 4.2.

Notation 1 ($X \in C$) *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 2 ($st(X, S)$) *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 1 (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 objects of C transit form s to s′ iff there exists a global state S such that R can make objects of C transit form s to s′ in S.*

Definition 2 (pre-transit local states) *Let R be a transition rule and C be a class of objects, then pre-transit local states of R for C, denoted $prels(R, C)$, is a 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 objects of C transit from s to s′.*

For example, if $st(\texttt{myInstance}, S)$ is *initial* then `R01` can make `myInstance` transit from *initial* to *started* in S and thus $prels(\texttt{R01}, \texttt{Resource})$ is { `initial` }. Note that a transition rule can make objects of more than one classes transit.

Notation 3 ($S[X/s]$) *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 a 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* $S[X_1/s_1, X_2/s_2, \dots]$*.*

Definition 3 (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* X' *is included in a set of pairs of an object and a local state,* Σ*, such that* 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* $dep_R(X, X', S)$*.*

Definition 4 (depending set) *Let* X *be an object,* R *be a transition rule, and* S *be a global state, then a 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 5 (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*.*

Let S_0 be the following global state:

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

$DS(\texttt{myEIP}, S_0) = \{ \texttt{myEIP::InsID}, \texttt{myInstance} \}$, 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*. Thus, 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; that is `myInstance`.

Note that when using the Cyclic Dependency Lemma for a transition rule R which can make objects of a class C transit, we can only focus on objects of C.

Definition 6 (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 a depending set of the same class as X in S, denoted $DSC(X, S)$, is defined as $DSC(X, S) = \{ X' \in C \mid X' \in DS(X, S) \}$*

In order to show no cyclic dependency, we should only check whether $DSC(X, S)$ does not include X itself. And when we find some object of class C whose local state is included in $prels(R, C)$, then we can assume there exists some object O of C whose local state is also included in $prels(R, C)$ and $DSC(O, S)$ includes no object whose local state is included in $prels(R, C)$; typicall $DSC(O, S)$ is empty.

Definition 7 (dependency chain) *Let $X_1, X_2, \ldots, X_n$ be objects and S be a global state, then a 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, $dc([\texttt{myEIP},\texttt{myEIP::InsID},\texttt{myInstance}], S_0)$.

Definition 8 (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. A directly depending set of the same class as X in S, denoted $DDSC(X, S)$, is defined as $\{ X' \mid \exists dc([X, X_1, \ldots, X_n, X'], S) \wedge X' \in C \wedge \forall i \in [1 \ldots n] : X_i \notin C \}$.*

When X and X' are objects of C, $X' \in DDSC(X)$ 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, $DDSC(\texttt{myEIP}, S_0) = \{ \texttt{myInstance} \}$ since there is a dependency chain $dc([\texttt{myEIP},\texttt{myEIP::InsID},\texttt{myInstance}], S_0)$.

How to Use Cyclic Dependency Lemma in Verification

Using the formalization of cyclic dependency explained above, the framework provides a predicate, `noCycle(S)`, which checks there is no cyclic dependency in a global state S. A template module, `CYCLEPRED`, together with a parameter module, `PRMCYCLE`, defines the predicate as follows:

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

  [State]
  op getAllObjInState : State -> SetOfObject

  -- DDSC means get Direct Depending Set of the same class.
  -- DDSC is required to have the following properties.
  --   (O \in DDSC(O,S)) = false .
  op DDSC : Object State -> SetOfObject

  -- DDSC means get Direct Depending Set of the same class.
  -- DDSC is required to have the following properties.
  --   (O \in DDSC(O,S)) = false .
  --   X \in DDSC(O,S) implies X \in DDSC(O,S)
  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
  pred noCycleStructure : State
  pred noCycleStructure : 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) .

  eq noCycleStructure(S)
     = noCycleStructure(getAllObjInState(S),empObj,S) .
```

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

Operator `getAllObjInState(`S`)` returns a set of all objects in S of the specific class we concerns. For the set of objects, predicate `noCycle` transitively visits objects in directly depending sets `DDSC(O,`S`)` and checks not to find any object already visited. Since `getAllObjInState` and `DDSC` are specific to each problem, the user of the framework should appropriately define them. If it is proved that `noCycle(S)` is an invariant, then the Cyclic Dependency Lemma can be used in verification to ensure that there is some object which does not depend on other objects and the rule can make it transit.

In addition, the framework provides a utility predicate, `noCycleStructure`, which processes the same manner using directly depending sets `DDSC`. Since `DDSC` represents overall dependency between objects in the class without concerning their local states, `noCycleStructure` checks no structural cyclic dependency. In other words, it checks static dependency while `noCycle` checks dynamic dependency. Since $DDSC(X,S)$ is a subset of $DDSC(X)$, if `noCycleStructure(S)` is an invariant then `noCycle` is also an invariant. It is often the case that a system is designed to have no static cyclic dependency in order to avoid complicate control of dynamic dependency. When verifying such system, the only thing the user of our framework should do is to put `noCycleStructure` into the initial condition of the system, which ensures that `noCycle` is an invariant unless any transition rule changes the system structure.

For example, for rule `R01` in Section 4.2, three actual parameter operators given to `CYCLEPRED` are defined as follows: properties.

```
module! STATECyclefuns {
  pr(STATE)

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

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

  op DDSCR01 : Resource State -> SetOfResource
  eq DDSCR01(RS,< SetRS,SetPR >)
     = DDSCR01(RS,< SetRS,SetPR >) .

  op DDSCR01 : Resource State -> SetOfResource
  eq DDSCR01(RS,< SetRS,SetPR >)
     = getRRSsOfPRs(SetRS,getPRsOfRS(SetPR,RS)) .
}
```

Here, `getPRsOfRS` returns all properties of a resource and `getRRSsOfPRs` returns all referred resources by a set of properties. Note that $DDSC(X)$ is defined without depending on any specific global state while function `DDSCR01` needs a global state argument because it also provides

the structural information of the cloud system. Template module CYCLEPRED is instantiated as follows:

```
extending(CYCLEPRED(
 STATECyclefuns {sort Object -> Resource,
          sort SetOfObject -> SetOfResource,
          op empObj -> empRS,
          op getAllObjInState -> getAllRSInState,
          op DDSC -> DDSCR01,
          op DDSC -> DDSCR01})
 * {op noCycle -> noRSCycle,
    op noCycleStructure -> noRSCycleStruct}
 )
```

noCycle and noCycleStructure are renamed as noRSCycle and noRSCycleStruct. Only we have to do is to put noRSCycleStruct(S) into the initial condition of the system and then we can use the Cyclic Dependency Lemma to assume the existence of an object to which R01 will be applied.

*** Usage for a lemma ***

Chapter 6

Verification Procedure of Leads-to Properties

The framework also provides an overall verification procedure for leads-to properties. It assists developers to systematically think and develop proof scores.

A typical property of an automated system setup operation, which we want to verify, is that the operation surely brings a cloud system to the state where all of its resources are started. We say "surely" to mean that the sytem always reaches some final state from any initial state. This kind of reachability is one of the most important properties of practical automation of cloud systems.

The initial and final states, represented as predicates $init(S)$ and $final(S)$, can be specified by equations in CafeOBJ as follows.

```
eq init(< SetRS,SetPR >)
   = wfs(< SetRS,SetPR >) and
     allRSInStates(SetRS,initial) and
     allPRInStates(SetPR,notready) .
eq wfs(< SetRS,SetPR >)
   = not (SetRS = empRS) and
     uniqRS(SetRS) and uniqPR(SetPR) and
     allPRHaveRS(SetPR,SetRS) and
     allPRHaveRRS(SetPR,SetRS) and
     noRSCycleStruct(< SetRS,SetPR >) .
eq final(< SetRS,SetPR >)
   = allRSInStates(SetRS,started) .
```

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`.

When automation is modeled as a state machine, reachability mentioned above is formalized as ($init$ `leads-to` $final$) which means that any transition sequence from any initial state always reaches some final state no matter what possible transition sequence is taken.

Lemma 6 (`leads-to` Property Lemma) *Let cont be a state predicate, inv be a conjunction of some state predicates, and m be a natural number function of a global state. If there exist cont, inv, and m such that the following six conditions hold where S′ means any possible next state*

of S, then (init `leads-to` $final$*) sufficiently holds [5]:*

$$\forall S: \quad init(S) \rightarrow cont(S) \tag{6.1}$$

$$\forall S, S': \quad (inv(S) \wedge cont(S) \wedge \neg final(S)) \rightarrow (cont(S') \vee final(S')) \tag{6.2}$$

$$\forall S, S': \quad (inv(S) \wedge cont(S) \wedge \neg final(S)) \rightarrow (m(S) > m(S')) \tag{6.3}$$

$$\forall S: \quad (inv(S) \wedge (cont(S) \vee final(S)) \wedge (m(S) = 0)) \rightarrow final(S) \tag{6.4}$$

$$\forall S: \quad init(S) \rightarrow inv(S) \tag{6.5}$$

$$\forall S, S': \quad inv(S) \rightarrow inv(S') \tag{6.6}$$

Here, *cont* represents whether the state machine continues to transit from the given state. Condition (6.1) means an initial state should be a continuing state, i.e. it should start transitions. Condition (6.2) means transitions continue until $final(S')$ holds. Condition (6.3) 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 (6.4) then ensures $final(S')$. Here, m is called a *state measuring function*[1]. When condition (6.5) and (6.6) hold, each state predicate included in *inv* is called an invariant. Note that all wfs conditions should be an invariant.

6.1 Procedure: Definition of Support Operators

Step 0-1: Define *cont*.
Since $cont(S)$ means that state S has at least one next state, it can be specified as follows using the search predicate of CafeOBJ .

```
eq cont(S) = (S =(*,1)=>+ S') .
```

Step 0-2: 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 $\sum_C \sum_{0 \leq k \leq n_C} \#st_C^k \times (n_c - k)$ where $\#st_C^k$ is the number of objects of class C whose local state is st_C^k. For the example show in Fig. 2.1, m can be defined as follows:

```
eq m(< SetRS,SetPR >)
   = (#ResourceInStates(initial,SetRS) * 1)
   + (#ResourceInStates(started,SetRS) * 0)
   + (#PropertyInStates(notready,SetPR) * 1)
   + (#PropertyInStates(ready,SetPR) * 0) .
```

[1]Researches on verification of liveness properties often assume fairness constraints to make state machines always reach desired states, whereas our lemma requires a properly decreasing function, m, which is strong enough for such reachability. Since cloud orchestration intentionally brings a cloud system to desired states, the specification usually designs straight forward behavior which typically results in existence of some state measuring function m.

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 state 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.

6.2 Procedure: Proof of Condition (6.1)

Step 1-0: Define a predicate to be proved.
Predicate `initcont` to represent condition (6.1) can be defined as follows:

```
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 (6.1), the global state consists of arbitrary constants every of which represents an arbitrary set of objects of each class. For the example show in Fig. 2.1, the most general case is as follows where `sRS` and `sPR` are arbitrary constants for a set of resources and properties respectively. This case is too general to judge whether the condition does or does not hold. Thus, no reduction occurs.

```
-- Case 1 of Condition (6.1)
reduce initcont(< sRS,sPR >) .
```

Step 1-2: Think which rule is firstly applied to an initial state.
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 cannot find the first applied rule, it means insufficient understanding of the specification. For the example show in Fig. 2.1, the first rule is `R01`.

Step 1-3: Split the general case into cases which collectively cover the general case and one of which matches to LHS of the first rule.
Since LHS of rule `R01` requires the global state to have at least one `initial` resource, the case is split into three more cases; no resource, at least one `initial` or `started` resource. In the following proof score, `trs`, `idRS`, and `sRS'` are arbitrary constants for a type, an identifier, and a set of resources respectively. Case 1.1 and 1.3 reduce to true because the antecedent of `initcont(S)`, i.e. `init(S)`, does not hold in those cases. Only Case 1.2 remains too general.

```
-- Case 1.1 of Condition (6.1)
eq sRS = empRS .
reduce initcont(< sRS,sPR >) .

-- Case 1.2 of Condition (6.1)
eq sRS = (res(trs,idRS,initial) sRS') .
reduce initcont(< sRS,sPR >) .
```

```
-- Case 1.3 of Condition (6.1)
eq sRS = (res(trs,idRS,started) sRS') .
reduce initcont(< sRS,sPR >) .
```

Step 1-4: Split the first rule 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 1.2 is split into two more cases; all properties are or are not `ready`. The Set Lemma ensures that these cases are represented as follows where only Case 1.2.2 remains too general.

```
-- Case 1.2.1 of Condition (6.1)
eq sRS = (res(trs,idRS,initial) sRS') .
eq allPROfRSInStates(sPR,idRS,ready) = true .
reduce initcont(< sRS,sPR >) .

-- Case 1.2.2 of Condition (6.1)
eq sRS = (res(trs,idRS,initial) sRS') .
eq sPR = (prop(tpr,idPR,notready,idRS,idRRS) sPR') .
reduce initcont(< sRS,sPR >) .
```

Step 1-5: When there is a dangling link, split the 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`. Thus, 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 and is typically rejected by a wfs (`allPRHaveRRS`). Case 1.2.2 is split into the following three cases where only Case 1.2.2.2 remains too general.

```
-- Case 1.2.2.1 of Condition (6.1)
eq sRS = (res(trs,idRS,initial) sRS') .
eq sPR = (prop(tpr,idPR,notready,idRS,idRRS) sPR') .
eq existRS(sRS',idRRS) = false .
reduce initcont(< sRS,sPR >) .

-- Case 1.2.2.2 of Condition (6.1)
eq sRS = (res(trs,idRS,initial) sRS') .
eq sPR = (prop(tpr,idPR,notready,idRS,idRRS) sPR') .
eq sRS' = (res(trs',idRRS,initial) sRS'') .
reduce initcont(< sRS,sPR >) .

-- Case 1.2.2.3 of Condition (6.1)
eq sRS = (res(trs,idRS,initial) sRS') .
eq sPR = (prop(tpr,idPR,notready,idRS,idRRS) sPR') .
eq sRS' = (res(trs',idRRS,started) sRS'') .
reduce initcont(< sRS,sPR >) .
```

Step 1-6: When falling in a cyclic situation, use the Cyclic Dependency Lemma.
Since `noRSCycle` is a wfs and resource `idRS` is `initial`, the Cyclic Dependency Lemma ensures there exists some `initial` resource X' such that all resources in `DDSCR01(R,S)` are

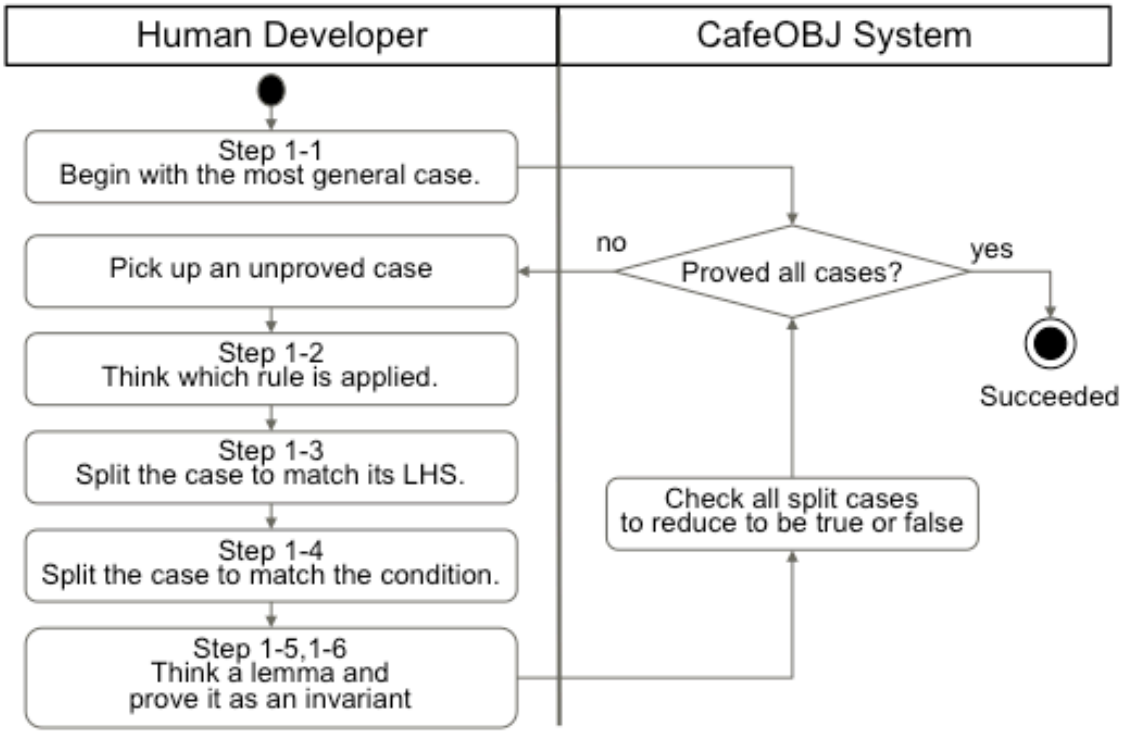

Figure 6.1: Verification Procedure for Condition (6.1)

`started`. Recalling that we chose `idRS` as an arbitrary `initial` resource in Step 1-3, we can assume that itself is such X' and can introduce the constraint as follows:

```
-- Case 1.2.2.2 of Condition (6.1)
eq sRS = (res(trs,idRS,initial) sRS') .
eq sPR = (prop(tpr,idPR,notready,idRS,idRRS) sPR') .
eq sRS' = (res(trs',idRRS,initial) sRS'') .
eq x' = res(trs,idRS,initial) .
reduce allRSInStates(DDSCR01(x',< sRS,sPR >), started)
    implies initcont(< sRS,sPR >) .
```

Since `DDSCR01` of resource `idRS` includes resource `idRRS` which is not `started`, the introduced constraint is not satisfied and so the whole case reduces to true.

Thus, all split cases reduce to true and condition (6.1) is proved. Figure 6.1 summarizes the procedure.

6.3 Procedure: Proof of Condition (6.2)

Step 2-0: Define a predicate to be proved.
Using the double negation idiom in Section 3.3, predicate `contcont` for condition (6.2) can be defined as follows [2]:

[2]When some next state `S'` of state `S` exists, it means `S` is a continuous state and `cont(S)` holds. Thus `cont(S)` can be omitted from `ccont(S,S')` and `mmes(S,S')`.

```
eq ccont(S,S')
   = inv(S) and not final(S)
     implies cont(S') or final(S') .
eq contcont(S)
   = not (S =(*,1)=>+ S' if CC suchThat
       not ((CC implies ccont(S,S')) == true)) .
```

Step 2-1: Begin with the cases each of which matches to LHS of each rules.
Since condition (6.2) 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 example show in Fig. 2.1, we can begin with two cases for two rules as follows, which are too general.

```
-- Case 2.R01 of Condition (6.2) for rule R01
reduce contcont(< (res(trs,idRS,initial) sRS), sPR >) .

-- Case 2.R02 of Condition (6.2) for rule R02
reduce contcont(< (res(trs,idRRS,started) sRS),
          (prop(tpr,idPR,notready,idRS,idRRS) sPR) >) .
```

Hereafter, we only explain the steps of the procedure by omitting to show the split case examples.

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

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

Step 2-4: Think which rule can be applied to the next state and repeat case splitting similarly as Step 1-3, 1-4, and 1-5 until all cases reduce to true .

Step 2-5: When falling in a cyclic situation, use the Cyclic Dependency Lemma.

6.4 Procedure: Proof of Condition (6.3)

Since the antecedent of condition (6.3) is equivalent to (6.2), the proof procedure of (6.3) is almost the same as of (6.2).

Step 3-0: Define a predicate to be proved.

```
eq mmes(S,S')
   = inv(S) and not final(S)
     implies m(S) > m(S') .
eq mesmes(S)
   = not (S =(*,1)=>+ S' if CC suchThat
        not ((CC implies mmes(S,S')) = true)) .
```

Step 3-1: Begin with the cases each of which matches to LHS of each rule.

Step 3-2: Split the most general case for a rule into cases where the condition of the rule does or does not hold.

6.5 Procedure: Proof of Condition (6.4)

Step 4-0: Define a predicate to be proved.

```
eq mesfinal(S)
   = (inv(S) and (cont(S) or final(S)) and m(S) = 0)
     implies final(S) .
```

Step 4-1: Instantiate a proved lemma.
The framework also provide a proved lemma such that:

```
eq base-lemma01(SetO,ST,SetST)
   = (allObjInStates(SetO,(ST SetST))
      and #ObjInStates(ST,SetO) = 0)
     implies allObjInStates(SetO,SetST) .
```

Recall that `allObjInStates` and `#ObjInStates` are instantiated as `allRSInStates` and `#ResourceInStates` respectively. Since there are only two kinds of local states of resources, `allRSInStates(SetRS,(initial started))` always holds and we can define a specific lemma as follows:

```
eq lemma(SetRS)
   = #ResourceInStates(ST,initial) = 0
     implies allRSInStates(SetRS,started) .
```

Step 4-2: Use a natural number axiom.

```
eq (N1 + N2 = 0) = (N1 = 0) and (N2 = 0) .
reduce lemma(sRS) implies mesfinal(< sRS, sPR >) .
```

6.6 Procedure: Proof of Condition (6.5) & (6.6)

Since (6.5) and (6.6) are conditions for invariants whose proof procedure is rather well known, here we only explain basic strategies:

- Prove each invariant separately but take care of dependency of invariants.
- Begin with the most general case and split it by thinking through meaning of the specification.
- Condition (6.6) can be proved for each rule similarly as Step 2-1, 2-2, and 2-3.
- Introduce appropriate lemmas and prove them using mathematical induction about a set of objects.
- Use provided lemmas, such as m2o-lemma07 explained in Section 5.3.2.

6.7 Using Mathematical Induction for Sets of Objects

When applying the framework to verification of the example transition rule set in Section 4.2, we need to split totally 35 cases and define four invariants and four lemmas. All lemmas are instantiated from proved lemmas.

Chapter 7

Applying the Framework to TOSCA Specifications

7.1 Structure Models of TOSCA Templates

We model a topology of a cloud application as a set of four kinds of objects corresponding to the four main kinds of elements of a topology; nodes, relationships, capabilities, and requirements. Each object has a type, an identifier, a (local) state and may have links to other objects. 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.

A type of nodes defines invocation rules of its operations. Each rule specifies when an operation can be invoked and how it changes the state of the node. A type of relationships also defines invocation rules of its operations. 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. As described in Section 2.4, type operations and their invocation rules should be defined by type architects. When an application architect defines a topology, a set of all type operations and a set of all invocation rules of referred node/relationship types collectively define behavior of the topology.

Let us use a typical example where four node types and three relationship types in Fig. 2.2 participate in automation of a setup operation. In this example, 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.

7.2 Behavior Models of TOSCA Templates

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

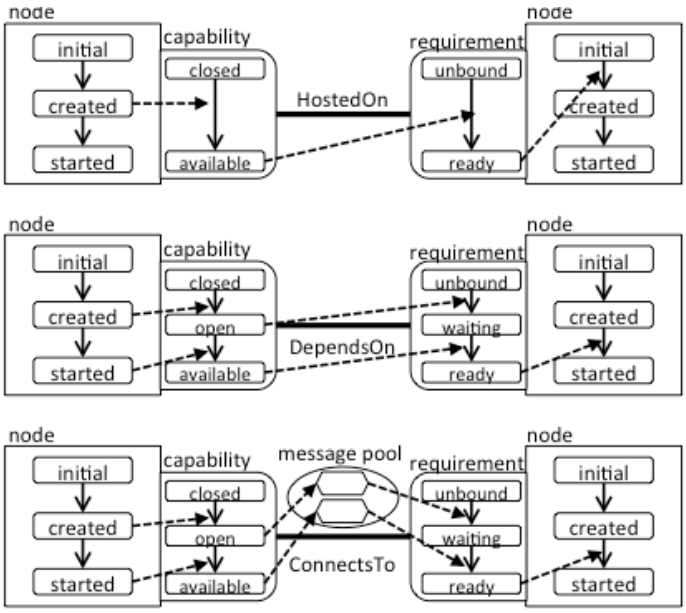

Figure 7.1: Typical Behavior of Relationship Types

its requirement. We also assume that types of capabilities and requirements are the same as relationships that link them in this example for the sake of simplicity.

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

Initial States: Every node is initially in a 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 become *ready* and changes the state from *initial* to *created*.
- *start* operation can be invoked if all of the requirements become *ready* and changes the 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 state of its capability from *closed* to *available*.
- *reqready* operation can be invoked if its capability is *available* and changes the 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 state of its capability from *closed* to *open*.
- *capavailable* operation can be invoked if the target node is *started* and changes the 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 state of its requirement from *unbound* to *waiting*.
- *reqready* operation can be invoked if its capability is *available* and changes the 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 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 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 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 state of its requirement from *waiting* to *ready*.

The model described in the previous section is specified by twelve transition rules two of which are for node operations, two are for operations of HostedOn relationship, and eight are for four operations of two relationship types. The followings show three of them for *create* and *start* operation of nodes (R01, R02) and *reqready* operation of ConnectsTo relationship (R12):

```
-- Create an initial node if all of its hostedOn requirements are ready.
ctrans [R01]:
   < (node(TND,IDND,initial) SetND), SetCP, SetRQ, SetRL, MP >
=> < (node(TND,IDND,created) SetND), SetCP, SetRQ, SetRL, MP >
   if allRQOfNDInStates(filterRQ(SetRQ,hostedOn),IDND,ready) .

-- Start a created node if all of its requirements are ready.
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) .

-- Let a waiting ConnectsTo requirement be ready
-- if there is an available message of the corresponding capability.
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 > .
```

Here, all terms staring with capital letters are pattern-matching variables. Since a blank character represents an associative, commutative, and idempotent operator to construct sets with the identity, `(ND1 ND2 ND3)` represents a set of nodes and `(ND SetND)` also represents a set of nodes when `NDn` are nodes and `SetND` is a set of nodes. Predicate `allRQOfNDInStates(SetRQ,IDND,ready)` checks whether every requirement in `SetRQ` is *ready* if the identifier of its node is `IDND`. `filterRQ(SetRQ,hostedOn)` is a subset of `SetRQ` which elements are HostedOn requirements. Note that `allRQOfNDInStates(SetRQ,IDND,ready)` always holds when node `IDND` has no requirements in `SetRQ`. `(avMsg(IDCP) MP)` means the message pool includes at least one available message of capability `IDCP`.

7.3 Simulation of TOSCA Templates

7.4 Verification of TOSCA Templates

A typical property of an automated system setup operation, which we want to verify, is that the operation surely brings a cloud application to the state where all of its component nodes are *started*. We say "surely" to mean total reachability, i.e. any transition sequence from any initial state always reaches some final state. Total reachability is one of the most important properties of practical automation of cloud applications.

The initial and final states are represented as predicates $init(S)$ and $final(S)$ that can be specified by equations in CafeOBJ as follows.

```
eq init(< SetND,SetCP,SetRQ,SetRL,MP >)
   = not (SetND = empND) and wfs(< SetND,SetCP,SetRQ,SetRL,MP >) and
     (MP = empMsg) and allNDInStates(SetND,initial) and
     allCPInStates(SetCP,closed) and allRQInStates(SetRQ,unbound) .
eq wfs(< SetND,SetCP,SetRQ,SetRL,MP >)
   = allCPHaveND(SetCP,SetND) and allRQHaveND(SetRQ,SetND) and
     allRLHaveCP(SetRL,SetCP) and allRLHaveRQ(SetRL,SetRQ) and
     allRQHaveRL(SetRQ,SetRL) and allRLNotInSameND(SetRL,SetCP,SetRQ) .
eq final(< SetND,SetCP,SetRQ,SetRL,MP >) = allNDInStates(SetND,started) .
...
eq allRLNotInSameND(empRL,SetCP,SetRQ) = true .
eq allRLNotInSameND((RL SetRL),SetCP,SetRQ)
   = (node(getCapability(SetCP,RL))
      = node(getRequirement(SetRQ,RL))) = false
     and allRLNotInSameND(SetRL,SetCP,SetRQ) .
```

Here, we omitted definitions of several predicates; `allNDInStates(SetND, initial)` means that every node in `SetND` is *initial*, `allCPHaveND(SetCP,SetND)` means that every capability in `SetCP` has its node in `SetND`, and so on. Note that predicate `wfs` (well-formed state) specifies conditions that should hold in not only initial states but also any reachable states.

We have proposed how to specify behavior of TOSCA topologies as state machines and use an example state machine consisting of twelve trans rules to verify that orchestrated operations always successfully complete [17]. For the example, we used about 800 cases to verify the six conditions defined in Chapter 6 and had to define 28 invariants and many lemmas. Here, we report how we apply the framework to the same problem and the result.

TOSCA's four classes of object (node, relationship, capability, and requirement) can be specified only by renaming template `OBJECTBASE`. Links between a capability and a node, a requirement and a node, and a relationship and a capability can be easily specified by using template `OBJLINKMANY2ONE`. Links between a relationship and a requirement can be specified by template `OBJLINKONE2ONE`. TOSCA distinguishes two kinds of relationships; a local relationship is between nodes in the same VM where a remote one is between nodes in different VMs. The capability and requirement of a remote relationship cannot directly refer each other and instead should communicate those local states using a messaging mechanism. Thus, we model a global state consisting of not only the sets of objects of four classes but also a message pool like as `< SetND,SetCP,SetRQ,SetRL,MP >`.

19 of 28 invariants can be defined only by renaming or combining predefined predicates. We need additional coding for the following three reasons:

- Five invariants check the consistency between a message and local states of object, e.g. if there is an available message then the corresponding capability should be available.
- An invariant checks the type consistency among relationships, capabilities, and requirement.
- Three invariants check other problem-specific constraints, e.g. every node should be hosted on exactly one VM node.

The invariants defined by predefined predicates can be proved by instantiated general lemmas of the framework. Thus, our efforts mainly focus on proving nine other invariants, which is structured work assisted by the verification procedure.

By using our framework, time and efforts to develop them is radically reduced although the number of required cases is essentially the same. 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 policy 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 (6.2)(3)(6) which should be proved for each of twelve transition rules.

Chapter 8

Related Work and Conclusion

8.1 Related Work

8.1.1 Formal Approach for Cloud Orchestration

Salaün, G., et al. [4, 14, 15] designed a system setup protocol and demonstrated to verify a liveness property of the protocol using their model checking method. 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.

8.1.2 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. 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 more general formalization for the domain of cloud orchestration and also provide a framework for developing specifications and their proofs.

8.2 Future Issues

While more than half of invariants and lemmas for the TOSCA specification can be easily defined by using predefined predicates and lemmas, extension of our framework is desired to reduce problem specific coding and proving. The general formalization for messaging mechanism and type system 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 applications. 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 scale-in/out 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 showing that `noCycle(S)` is an invariant, 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.

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 a general model and a procedure for verifying leads-to properties of declarative cloud orchestration. The 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.

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.

All CafeOBJ codes of the framework and example proof scores 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] Chef Software Inc. Chef |IT Automation for speed and awesomeness. https://www.chef.io/chef/, Accessed: 2016-06-10.

[4] 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.

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

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

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

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

[9] Internet Engineering Task Force (IETF). The JavaScript Object Notation (JSON) Data Interchange Format, 2014. https://tools.ietf.org/html/rfc7159, Accessed: 2016-03-26.

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

[11] 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.

[12] Puppet. Puppet - The shortest path to better software. https://puppet.com/, Accessed: 2016-06-10.

[13] Red Hat Inc. Ansible is Simple IT Automation. http://www.ansible.com/, Accessed: 2016-06-10.

[14] 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.

[15] 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.

[16] The OpenStack project. Heat - OpenStack. `https://wiki.openstack.org/wiki/Heat`, Accessed: 2016-06-10.

[17] Hiroyuki Yoshida, Kazuhiro Ogata, and Kokichi Futatsugi. Formalization and verification of declarative cloud orchestration. In *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings*, LNCS 9407, pages 33–49. Springer, 2015.