Formalization and Verification of Declarative Cloud Orchestration*

Hiroyuki Yoshida[1], Kazuhiro Ogata[1,2], and Kokichi Futatsugi[2]

[1] School of Information Science
[2] Research Center for Software Verification (RCSV)
Japan Advanced Institute of Science and Technology (JAIST)
1-1 Asahidai, Nomi, Ishikawa 923-1292, Japan

Abstract. Automation of cloud system operations, for example, which set up, monitor, and back up such systems, is called Cloud Orchestration. A standard specification language, TOSCA (Topology and Orchestration Specification for Cloud Applications), has been proposed to define topologies of cloud applications. A topology is a static structure of resources, such as VMs and software components, and a TOSCA conforming tool is expected to automate system operations based on the topologies. The current TOSCA standard, however, does not yet explicitly provide any way to formally define behavior of a topology (how to automate a topology). This paper proposes how to specify behavior of TOSCA topologies as state transition systems and to verify that orchestrated operations always successfully complete by proving the transition systems enjoys leads-to (a class of liveness) properties. We report on a case study in which we have specified and verified a setup operation to demonstrate the feasibility and usefulness of the proposed solution.

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.

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 management of cloud resources is called *resource orchestration*, or *cloud orchestration*.

* This work was supported in part by Grant-in-Aid for Scientific Research (S) 23220002 from Japan Society for the Promotion of Science (JSPS).

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* [10]. CloudFormation can manage resources provided by the IaaS platform of AWS, such as VMs, block storages (EBS), and load balancers (ELB). CloudFormation automatically sets up these resources according to declaratively defined 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 Apache Httpd package, creating configuration files, copying HTML 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* [11], *Chef* [4], and *Ansible* [2].

Currently 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 first 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* [7] that can be used to describe the structure of any type of resources. The resource structure is called a *topology* and a TOSCA tool is expected to automate system operations based on resource dependencies declaratively defined 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.

The contributions of this paper are as follows; (1) modeling and specifying automation of TOSCA topologies as state transition systems in CafeOBJ [3], an algebraic specification and verification system, and (2) verifying that the specification enjoys a desired property of surely reaching a goal state.

The rest of the paper is organized as follows. Section 2 briefly introduces OASIS TOSCA. Section 3 describes a model of the TOSCA topology and automation. Section 4 describes how we specify the model in CafeOBJ. Section 5 presents proof scores that verify a liveness property of a setup operation as an example. Section 6 explains related work and future issues.

2 TOSCA: Topology and Orchestration Specification for Cloud Application

TOSCA is a language to define a *service template* for a cloud application. 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. 1 shows a typical example of topology that consists of nine

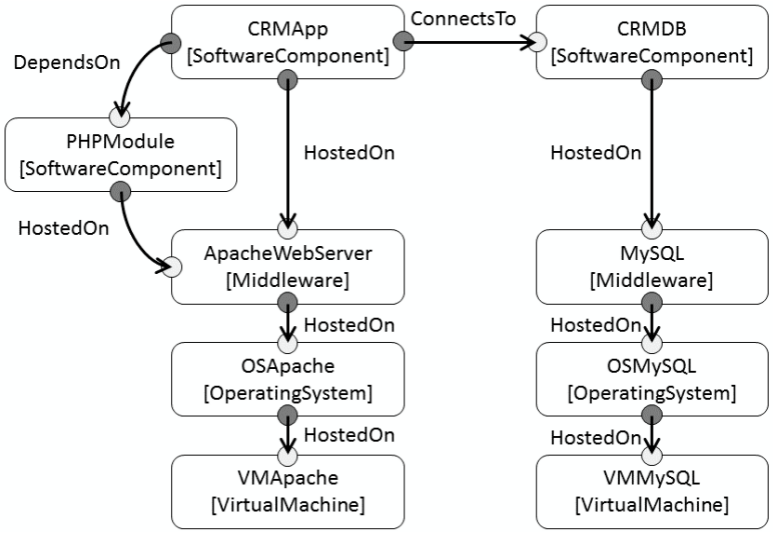

Fig. 1. An Example of TOSCA topology

nodes and nine relationships. White circles represent capabilities and black ones are requirements.

The current version of TOSCA is an XML-based language. The following is a part of the topology template of Fig. 1.

```
<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>
```

Each node, relationship, capability, and requirement has a type. In this figure, node types are Virtual Machine (VM), Operating System (OS), Middleware (MW), and Software Component (SC) and relationship types are HostedOn, DependsOn, and ConnectsTo. 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.

3 Model of Automation of Topologies

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, 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. 1 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.

On the other hand, behavior of relationship types usually varies according to their nature; they may be in the IaaS layer or in the inside of VM layer, "local" or "remote", "immediate" or "await". Three relationship types of this example typically cover the variation. A HostedOn relationship is one between resources in the IaaS layer. It is "immediate", i.e. it can be established as soon as the target node is created. Each of DependsOn and ConnectsTo relationships is between resources inside of VMs and is "await", i.e. it should wait for the target node to be started. A DependsOn relationship is "local" in the same VM, while a ConnectsTo is "remote" to a different VM and should use some messages to notice the states of its capability to its requirement. We 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. 2. 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*.

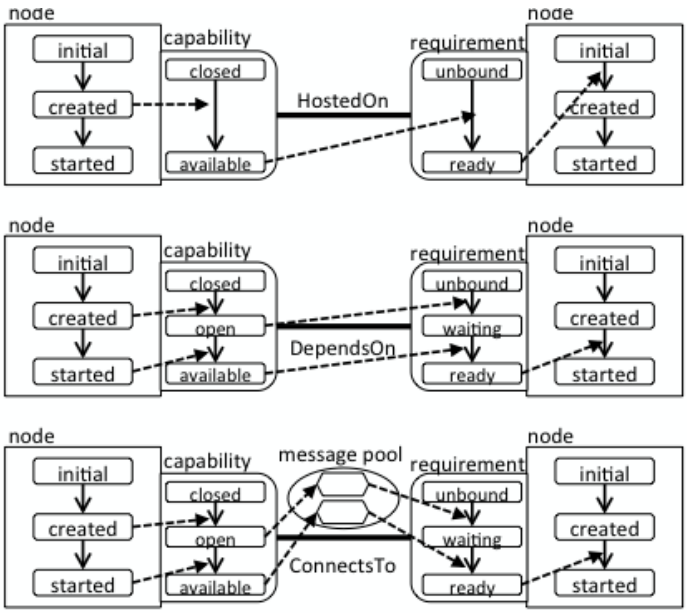

Fig. 2. Typical Behavior of Relationship Types

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*.

4 CafeOBJ Specification of Model

CafeOBJ [3] is a formal specification language that inherits many advanced functionalities from OBJ [8] and OBJ3 [9] algebraic specification language. 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.

Let l and r be terms of the same sort including a set of variables X, and let c be a term of sort $Bool$, then $(\forall X)(l = r \texttt{ if } c)$ is called a (conditional) equation. Let $State$ be a sort of states, l and r be terms of sort $State$ including a set of variables X, and let c be a term of sort $Bool$, then $(\forall X)(l \rightarrow r \texttt{ if } c)$ is called a transition rule. Let St be an sorted quotient term algebra of $State$ by equality, and let Tr be a set of transitions on the states where $Tr \subseteq St \times St$. A transition sequence is a sequence of states $(S_0, S_1, ...)$ where each adjacent pair $(S_i, S_{i+1}) \in Tr$.

A model of automation of a topology is specified as a transition system in CafeOBJ. A node is represented as a term `node(type,idND,state)` whose sort is `Node` where `idND` is its identifier. Similarly, a capability as `cap(type,idCP, state,idND)`, a requirement as `req(type,idRQ,state,idND)`, and a relationship as `rel(type,idRL,idCP,idRQ)` where `idCP`, `idRQ`, and `idRL` are identifiers of the capability, requirement, and relationship, respectively. In order to specify the whole application, let a global state `S` be a tuple `< nodes,caps,reqs,rels,mp >` whose sort is `State` where `nodes`, `caps`, `reqs`, and `rels` are sets of nodes, capabilities, requirements, and relationships respectively and `mp` is a message pool.

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`. All CafeOBJ codes of this example can be downloaded at `http://goo.gl/s9fJXq`.

5 Verification of Setup Operation

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.

When automation is modeled as a transition system, total reachability is formalized as ($init$ `leads-to` $final$) which means that any transition sequence from any initial state reaches some final state no matter what possible transition sequence is taken. Let *cont* be a state predicate representing whether the transition system continues to transit, *inv* be a conjunction of some state predicates, and m be a natural number function of a global state. Then the following six conditions are sufficient for proving that ($init$ `leads-to` $final$) holds [6].

(1) $(\forall s \in St)\ (init(s)$ `implies` $cont(s))$
(2) $(\forall (s, s') \in Tr)$
$((inv(s)$ `and` $cont(s)$ `and` (`not` $final(s)))$ `implies` $(cont(s')$ `or` $final(s')))$
(3) $(\forall (s, s') \in Tr)$
$((inv(s)$ `and` $cont(s)$ `and` (`not` $final(s)))$ `implies` $(m(s) > m(s')))$
(4) $(\forall s \in St)$
$((inv(s)$ `and` $(cont(s)$ `or` $final(s))$`and` $(m(s) = 0))$ `implies` $final(s))$
(5) $(\forall s \in St)\ (init(s)$ `implies` $inv(s))$
(6) $(\forall (s, s') \in Tr)\ (inv(s)$ `implies` $inv(s'))$

When condition (5) and (6) hold, each state predicate included in *inv* is called an invariant. And m is called a state measuring function.

Condition (1) means an initial state should be a continuing state, i.e. it should start transitions. Conditions (2) means transitions continue until $final(s')$ holds. Condition (3) implies that $m(s)$ keeps to decrease properly while $final(s)$ does not hold, but $m(s)$ is a natural number and should stop to decrease in finite steps, and should get to the state s' with $((cont(s')$ `or` $final(s'))$ `and` $(m(s') = 0))$. Condition (4) asserts that it implies $final(s')$.

CafeOBJ provides a built-in search predicate (s `=(*,1)=+` s') which returns `true` for state s if there exists state $s' \in St$ such that $(s, s') \in Tr$. Since $cont(s)$ means that state s has at least one next state, it can be specified as follows.

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

As to state measuring function $m(s)$, we should find a natural number function that properly decreases in transitions. For this purpose, we intentionally designed the transition system where every transition rule changes local states of at least one objects. Function m can be the weighted sum of counting local states of three sorts of objects as follows.

```
eq m(< SetND,SetCP,SetRQ,SetRL,MP >)
   = (( (#NodeInStates(initial,SetND) * 2)
   +     #NodeInStates(created,SetND))
```

```
    +  ( (#CapabilityInStates(closed,SetCP) * 2)
    +     #CapabilityInStates(open,SetCP)))
    +  ( (#RequirementInStates(unbound,SetRQ) * 2)
    +     #RequirementInStates(waiting,SetRQ)) .
```

For example, when rule `R01` is applied, `(#NodeInStates(initial,SetND) * 2)` decreases by two while `#NodeInStates(created,SetND)` increases by one and thus $m(s') = m(s) - 1$ holds. When $m(s)$ becomes 0, all nodes are not *initial* or *created*, i.e. are *started* which means the state is final. Defining $m(s)$ as above makes conditions (3) and (4) naturally hold.

The rest of this section presents proofs for these conditions. Although it is an interactive process, it is based on very systematic way of thinking and achieves structural and deep understanding of models, which is required in order to develop trusted systems.

5.1 Proof Score for Condition (1)

One of interesting features of CafeOBJ is that theorems to be proved and their proofs are written in the same executable specification language. A proof written in CafeOBJ is called a proof score.

The proof score for condition (1) begins with defining a theorem to be proved, i.e. $initcont(S)$.

```
  eq initcont(S) = init(S) implies cont(S) .
```

The most general case we can consider is when state `S` is `< sND,sCP,sRQ,sRL,mp >` where `sND` is an arbitrary constant representing a set of nodes and similarly so `sCP`, `sRQ`, `sRL`, and `mp`. But this case is too general for CafeOBJ to determine whether $initcont(S)$ does or does not hold.

Thinking through meanings of the model, we know that rule `R01` is firstly applicable to an initial state. `R01` can be applied when there is at least one *initial* node and all of its HostedOn requirements are *ready*. The proof score is hence split into five cases; there is no node, at least one *created* node, one *started* node, one *initial* node whose HostedOn requirements are *ready*, and one *initial* node one of whose HostedOn requirements is not *ready*. Let `tnd` be an arbitrary constant representing any type of node and let `sND'` be any set of nodes, then the proof score for the first four cases is as follows:

```
-- Case 1: There is no node.
  eq sND = empND .
  reduce initcont(< sND, sCP, sRQ, sRL, mp >) .
  --> to be true because init(S) does not hold.
...
-- Case 2: There is at least one created node.
  eq sND = (node(tnd,idND,created) sND') .
  reduce initcont(< sND, sCP, sRQ, sRL, mp >) .
  --> to be true because init(S) does not hold.
...
-- Case 3: There is at least one started node.
```

```
  eq sND = (node(tnd,idND,started) sND') .
  reduce initcont(< sND, sCP, sRQ, sRL, mp >) .
  --> to be true because init(S) does not hold.
...
-- Case 4: There is at least one initial node
--         all of whose HostedOn requirements are ready.
  eq allRQOfNDInStates(filterRQ(sRQ,hostedOn),idND,ready) = true .
  eq sND = (node(tnd,idND,initial) sND') .
  reduce initcont(< sND, sCP, sRQ, sRL, mp >) .
  --> to be true because cont(S) holds.
```

5.2 Cyclic Dependency

The fifth case requires more consideration.

When the *initial* node has one requirement that is not *ready*, there should be another node that has the corresponding capability. According to the state of such node, this case is split into four more cases. However, if at least one of its requirements is not *ready*, the case falls into a cyclic situation and the case splitting becomes endless.

Thinking through meanings of the model, we know that dependency of nodes should not be cyclic. We need to specify that a global state does not include any cyclic dependency, which is represented by state predicate $noCycleInState(S)$ specified as follows:

```
  eq noCycleInState(S) = noCycle(getAllNodeInState(S),empND,S) .
  eq noCycle(empND,V,S) = true .
  eq noCycle((ND SetND),V,S)
     = (not (state(ND) = initial) or
         if ND \in V then false else noCycle(DDS(ND,S),(ND V),S) fi)
       and noCycle(SetND,V,S) .
```

Here, `DDS(ND,S)`, *Directly Depending Set* of node `ND` in global state `S`, is a set of nodes whose local states are *initial* and on which `ND` is hosted. For example, let `S` be an initial state of the topology of Fig. 1, then `DDS(CRMApp,S) = (ApacheWebServer)` whereas `DDS(VMApache,S) = empND`. Predicate `noCycle(SetND, V,S)` traverses all transitive closures of `DDS` of all nodes in `SetND` and checks whether it does not reach some already visited node in `V`. In order to ensure that the acyclic property holds for any reachable states, $noCycleInState(S)$ should be included in $init(S)$ and also in $inv(S)$ and condition (6) should be proved.

Theorem: For any node `N` in state `S`, if $noCycleInState$(`S`) holds and the local state of `N` is *initial*, then there exists node `N'` in `S` such that the local state of `N'` is also *initial* and `DDS(N',S)` is empty.

Proof: If there is no such `N'` then `noCycle` will reach some already visited node because it traverses finite number of nodes. □

Let us return to the proof score of $initcont(S)$. When there is an *initial* node in state `S` and $noCycleInState$(`S`) holds, the theorem above allows us to assume that `DDS` of the node is empty. When the node has at least one requirement that is not *ready*, there is another node that has the corresponding capability and there are three more cases as follows:

```
-- When there is at least one initial node whose DDS is empty,
--      and at least one of whose HostedOn requirements is not ready,
-- there should be another node that has the corresponding capability.
  eq sND = (node(tnd,idND,initial) sND') .
  eq sRQ = (req(hostedOn,idRQ,srq,idND) sRQ') .
  eq (srq \in (unbound waiting)) = true .
  eq sRL = (rel(hostedOn,idRL,idCP,idRQ) sRL') .
  eq sCP = (cap(hostedOn,idCP,scp,idND1) sCP') .
-- Case 5: The corresponding node is created.
  eq sND' = (node(tnd',idND1,created) sND'') .
  reduce initcont(< sND, sCP, sRQ, sRL, mp >) .
  --> to be true because init(S) does not hold.
...
-- Case 6: The corresponding node is started.
  eq sND' = (node(tnd',idND1,started) sND'') .
  reduce initcont(< sND, sCP, sRQ, sRL, mp >) .
  --> to be true because init(S) does not hold.
...
-- Case 7: The corresponding node is initial.
  eq sND' = (node(tnd',idND1,initial) sND'') .
  reduce (DDS(node(tnd,idND,initial),< sND,sCP,sRQ,sRL,mp >) = empND).
  --> to be false, which is a contradiction,
  --> i.e. this is not a reachable state.
```

Thereby, $initcont(S)$ holds for all cases that are collaboratively exhaustive. Note that we do not explain several irrelevant cases such as inconsistent types, no corresponding capability, and relationship between the same node, because in such cases $wfs(S)$ does not hold and thus $initcont(S)$ holds.

5.3 Proof Score for Condition (2)

The proof score for condition (2) begins with defining a theorem to be proved, i.e. *contcont*.

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

This uses an idiom of a built-in search predicate, `=(*,1)=>+ _ if _ suchThat`. Given a global state `S`, `contcont(S,SS,CC)` searches all possible transitions from `S` while binding variable `SS` to each next state and checking condition (2) . It holds if and only if condition (2) holds for all such next states.

The most general case, `S = < sND,sCP,sRQ,sRL,mp >`, is too general because there is no rule to match with it. Thus the proof score should be split into twelve cases in each of which state `S` is as specific as each transition rule matches with it. This means that condition (2) (also (3) and (6)) can be proved rule by rule. For example, in the case of rule `R01`, state `S` should be as specific as its left-hand-side, `< (node(tnd,idND,initial) sND),sCP,sRQ,sRL,mp >`.

This situation is very instructive for us because we can find that the next state includes a *created* node and can expect that rule `R02` will be applicable. The proof score of rule `R01` is split into three cases; (1) the condition of `R01` does not hold, (2) it holds and the condition of `R02` holds, and (3) it does not hold. The proof score of the first two cases is as follows:

```
-- Case 1: The condition of R01 does not hold for S.
  eq allRQOfNDInStates(filterRQ(sRQ,hostedOn),idND,ready) = false .
  reduce contcont(< (node(tnd,idND,initial) sND),sCP,sRQ,sRL,mp >,SS,CC).
  --> to be true because cont(S) does not hold.
...
-- Case 2: The condition of R01 holds for S and
--         the condition of R02 holds for SS.
  eq allRQOfNDInStates(filterRQ(sRQ,hostedOn),idND,ready) = true .
  eq allRQOfNDInStates(sRQ,idND,ready) = true .
  reduce contcont(< (node(tnd,idND,initial) sND),sCP,sRQ,sRL,mp >,SS,CC).
  --> to be true because cont(SS) holds.
```

The last case means that the *initial* node in `S` becomes *created* in `SS` but at least one of its requirements is not *ready* (and is not HostedOn). There are thirty-six such cases, (2 relationship types, not HostedOn) $\times$ (2 requirement states, not *ready*) $\times$ (3 states of the corresponding capability) $\times$ (3 states of the corresponding node). Half of them are not reachable states because *reqwaiting* operation is never invoked when the source node is *initial* and thus the requirement should not be *waiting* in `S`. This property is required to be proved as an invariant. In each of other sixteen cases, one of twelve transition rules can be applicable and so $cont(SS)$ holds. For example, if a DependsOn requirement is *unbound* and the corresponding capability is *open*, then *reqwaiting* operation can be invoked in `SS` because the source node becomes *created*.

Remaining two cases are where a DependsOn or ConnectsTo requirement is *unbound*, the corresponding capability is *closed*, and its node is *initial*. Again we use the cyclic dependency theorem to assume that there is *initial* node `X` where `DDS(X,S)` is empty. Then, we repeat similar systematic case splitting as describe above, but this time we can reject another *initial* node.

5.4 Proof Scores for Condition (3), (4), (5) and (6)

As mentioned above, condition (3) can also be proved rule by rule. The following is a piece of the proof score for rule `R01`, in which a theorem of natural numbers is required:

```
  eq mesmes(S,SS,CC)
     = not (S =(*,1)=>+ SS if CC suchThat
            not ((CC then
                  (inv(S) and cont(S) and not final(S)
                   implies m(S) > m(SS))) == true){ ... }) .

-- A theorem of natural numbers
-- s(N) is a successor of N, i.e. s(N) = N + 1.
```

```
  eq (s(N) > N) = true .
...
-- Case for R01:
  reduce mesmes(< (node(tnd,idND,initial) sND),sCP,sRQ,sRL,mp >,SS,CC) .
```

For condition (4), we need a lemma such that if the number of *created* or *initial* nodes is zero, then all nodes are *started*. This lemma can be proved using mathematical induction about a set of nodes. The following is a part of the proof score for condition (4), where it also requires another natural number theorem.

```
-- Another theorem of natural numbers
  eq (N1 + N2 = 0) = (N1 = 0) and (N2 = 0) .
  eq lemma(SetND)
     = ((#NodeInStates(created,SetND) = 0) and
         #NodeInStates(initial,SetND) = 0)
       implies allNDInStates(SetND,started) .
  reduce lemma(sND) and (m(< sND,sCP,sRQ,sRL,mp >) = 0)
         implies final(< sND,sCP,sRQ,sRL,mp >) .
```

In order to prove conditions (1) and (2), we need more than ten invariants including $noCycleInState(S)$. Theorems to be proved are defined as follows:

```
  eq initinv(S) = init(S) implies inv(S) .
  eq invinv(S,SS,CC)
     = not (S =(*,1)=>+ SS if CC suchThat
            not ((CC then
                  (inv(S) implies inv(SS))) == true){ ... }) .
```

Similarly as described above, we can prove them using systematic case splitting and mathematical induction.

6 Related Work and Conclusion

Related Work

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 the formal specification of behavior of types and show that it can be used for verification.

CloudFormation and OpenStack Heat can manage resources on the IaaS layer, however, they support to manage dependencies between resources in VMs. For example, suppose a software component(SC_1) on a VM(VM_1) can be activated only after waiting for activation of another component(SC_2) on another VM(VM_2), CloudFormation requires a pair of special purpose resources, namely, *WaitCondition* and *WaitConditionHandle*. VM_1 should be declared to depend on the WaitCondition resource. The corresponding WaitConditionHandle resource provides a URL that should be passed to the script for initializing VM_2. When

SC_2 is successfully activated, the script sends a success signal to the URL, which causes the WaitCondition become active and then creation of dependent VM_1 starts. This style of management includes several problems. Firstly, it forces complicated and troublesome coding of operations. Secondly, although only SC_1 should wait for SC_2, all other components on VM_1 are also forced to wait. This causes unnecessary slowdown of system creation. Thirdly, it tends to make cyclic dependencies. Suppose SC_2 should also wait for another component SC_3 on VM_1. Although the dependency among components, SC_1, SC_2, and SC_3 is acyclic, the dependency between VMs is cyclic. This may be solved by splitting VM_1 to two VMs, one is for SC_1 and another is for SC_3, but it causes increased cost and delayed creation. Our formalization can manage any types of resources and solve this kind of problems in a smarter way because it can manage finer grained dependencies, which is shown as invocation rules described in Section 3.

Salaün, G., et al. [5, 12, 13] designed a system setup protocol and demonstrated to verify a liveness property of the protocol using their model checking method. Although their setup protocol is essentially the same as behavior of our example topology in this paper, there are two main differences. Firstly, their protocol is based on a specific implementation which challenges distributed management of cloud resources while current popular implementations, e.g. CloudFormation, implement centralized management. On the other hand, our model is rather abstract without assuming distributed or centralized implementations. Secondly, they used model checking while we use theorem proving. They checked about 150 different models of system including from four to fifteen components in which from 1.4 thousand to 1.4 million transitions are generated and checked. They found a bug of their specification because checked models fortunately included error cases. The model checking method can verify correctness of checked models and so they should include all boundary cases. In our formalization, the specification itself is verified by interactive theorem proving in which all boundary cases are necessary in consideration in a systematic way. It achieves structural and deep understanding that is required to develop trusted systems.

Future Issues

TOSCA supports type inheritance of any elements such as nodes, relationships, capabilities, and requirements. When a new type is defined and inherits a part of behavior from some existing type, it is desired that the corresponding part of the existing proof can be reused and only extended part of specification needs to be verified. While transition rules shown in this paper directly use type literals such as `hostedOn` or `connectsTo`, it is required to introduce some mechanism to use the rules for inherited types. The next version of TOSCA will define a set of standard types and we will introduce some inheritance mechanism to reuse proof scores of the standard types, which will significantly reduce efforts of proving behavior of topologies using inherited types.

The current version of TOSCA does not manage operation failures and it focuses on declaratively defining expected configurations of cloud applications. In many of failure cases, it is desired to roll back to the initial states, which

does not depend on correctness of topologies but on correctness of implementation of automation tools. A possible extension of TOSCA may be to define alternative configurations in failure cases, which we think we can easily extend our formalization to handle.

Conclusion

TOSCA topologies and automation are modeled, formalized, and verified with theorem proving. The specification and verification are demonstrated to prove total reachability of a typical set of relationship types. The proved specification is in a high abstraction level without depending on implementations of distributed or centralized managements, however, it provides a smarter solution than that of the most popular implementation. A related work proved similar problem by their model checking method, in which many of finite-state systems were checked. We use an interactive theorem proving method and verify applications of unlimited number of states in a significantly systematic way. Several general predicates and theorems are presented to be usable for common problems such as Cyclic Dependency.

References

1. Amazon Web Services: AWS CloudFormation: Configuration Management & Cloud Orchestration `http://aws.amazon.com/cloudformation/`, accessed: 2015-04-03
2. Ansible: Ansible is Simple IT Automation `http://www.ansible.com/home/`, accessed: 2015-04-03
3. CafeOBJ: http://www.ldl.jaist.ac.jp/cafeobj/ (2014)
4. Chef Software: IT automation for speed and awesomeness `https://www.chef.io/chef/`, accessed: 2015-04-03
5. Etchevers, X., Coupaye, T., Boyer, F., Palma, N.D.: Self-configuration of distributed applications in the cloud. In: IEEE CLOUD. pp. 668–675 (2011)
6. Futatsugi, K.: Generate & check method for verifying transition systems in cafeobj. In: Software, Services, and Systems. pp. 171–192 (2015)
7. 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: 2015-04-03
8. OBJ: http://cseweb.ucsd.edu/ goguen/sys/obj.html (2003)
9. OBJ3: http://www.kindsoftware.com/products/opensource/obj3/ (2005)
10. OpenStack.org: Heat - OpenStack Orchestration `https://wiki.openstack.org/wiki/Heat/`, accessed: 2015-04-03
11. Puppet Labs: IT Automation Software for System Administrators `https://puppetlabs.com/`, accessed: 2015-04-03
12. Salaün, G., Boyer, F., Coupaye, T., Palma, N.D., Etchevers, X., Gruber, O.: An experience report on the verification of autonomic protocols in the cloud. ISSE 9(2), 105–117 (2013)
13. Salaün, G., Etchevers, X., Palma, N.D., Boyer, F., Coupaye, T.: Verification of a self-configuration protocol for distributed applications in the cloud. In: Assurances for Self-Adaptive Systems. pp. 60–79 (2013)