Cloudscape: Language Support to Coordinate and Control Distributed Applications in the Cloud

Andi Bejleri*
Imperial College London
ab406@doc.ic.ac.uk

Andrew Farrell Patrick Goldsack
*HP Labs, Bristol
andrew.farrell@hp.com patrick.goldsack@hp.com

Abstract

Cloud Computing is an innovative computing proposal, which key feature is the ease and effectiveness of providing a service. There are a number of challenges that a management system for the Cloud will need to address including: scale, reliability (fault-handling and high availability), security and service heterogeneity, to achieve effectiveness.

This paper proposes an agent-oriented language, called CLOUDSCAPE, to address coordination and control of components in a distributed computation to provide reliability and scalability of service in the context of the Cloud. Agents are modeled as objects extended with transitions and dependencies to describe the lifecycle state machines of components and constraints between lifecycle states. The problem context is further extended with component failure and dynamic addition of new components. The practical utility and effectiveness of this system is illustrated through a series of real-world examples. We then define a formal model of the language and prove that the operational semantics of the language holds a linear consistent shared memory property.

1. Introduction

Cloud Computing is an innovative computing proposal that has emerged from technological developments of the last decade in computing, storage and networking. A key feature of this proposal is the ease and effectiveness of providing a service. While ease to provide a service is achieved using the web, effectiveness, including: scale, reliability (fault-handling and high availability), security and service heterogeneity, is addressed by a management system dealing with a series of challenges.

This paper studies *coordination* and *control* of components in a distributed computation in the context of the Cloud, providing reliability and scalability of service. In our study, a component is a set of functions written in a mainstream language, representing real world artifacts, e.g., a service in web services or a task in parallel algorithms and scientific computations. Components of a distributed application define *dependencies at different states of their lifecycles*— the sequence of states describing how components are deployed, run and destroyed. For example, in a basic Client-Server program [6], the running order of the two components is defined, citing the Java tutorial [6], as:

> *"When you start the client program, the server should already be running and listening to the port, waiting for a client to request a connection."*

Unfortunately, a violation of the constraint on the state of server would generate a run-time exception in the client code, and the computation will be established manually by restarting the client, as explained in the tutorial:

> *"If you are too quick, you might start the client before the server has a chance to initialize itself and begin listening on the port. If this happens, you will see a stack trace from the client. If this happens, just restart the client."*

The approach presented in the tutorial of using a human entity to coordinate the running of components cannot be applied to the Cloud. With millions of different service instances on roughly an order of magnitude more virtual machines running on the Cloud, coordinating manually the components of every distributed computation becomes intractable. The problem becomes more acute in presence of failure, coming from components logic or hardware, that affects the normal lifecycle of components. Thus, we would like a language that answers this research question:

> *How can programmers specify a management system that describes components lifecycle and dependencies between the lifecycles' states in a distributed computation and restores components normal lifecycle in case of failure?*

The solution proposed in this paper is an agent-oriented language CLOUDSCAPE, where an agent defines the active entity that (1) controls the execution of a component and (2) communicates with other agents to coordinate the execution of the distributed computation in ensemble. Agents are modeled as objects extended with transitions, dependencies and non-deterministic update. An *object* describes the state diagram of a component lifecycle and a *dependency* describes a causality between the lifecycle states of two components (source, target). Objects model behavior through the associated *transitions* that perform the change of component's lifecycle state by invoking components code. While methods in OO languages are a block of statements, CLOUDSCAPE transitions are a block of statements guarded by a predicate. A transition depending on the state of a second object (source) takes place only when that object satisfies the desired state. This relation is represented through CLOUDSCAPE dependencies, guarding the behavior of transitions at runtime. Non-deterministic update is used to describe scenarios where state can change normally to the next one, in accordance with the logic of the component, or exceptionally to restore the normal lifecycle in case of failure.

Another problem is to coordinate and control new components added at run-time typically to rescale the service due to load. For example, in the Load Balancer example, the Load Balancer adds new Web Servers into the session to handle greater request of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AGERE '11 24 October, Portland.
Copyright © 2011 ACM [to be supplied]. . . $10.00
Reprinted from AGERE '11, Proceedings of the ACM DL Actors and Agents Reloaded, 24 October, Portland., pp. 1–12.

load while maintaining reasonable user response time. Our formal model needs to address also a second research question:

How can programmers specify a management system that coordinates and controls components added dynamically?

The solution to this problem is providing CLOUDSCAPE with the feature of adding new agents dynamically; that is, adding new objects and dependencies from the body of transitions.

Our solution to both the questions follows a distributed approach, where agents themselves structure and share the control on components. This contrasts the centralised approach, known as the workflow approach to the SOA community, of actual management systems such as ControlTier [3], Capistrano [1] and HP Server Automation [5], where a central, monolithic unit controls all the components of an application. Our distributed approach suits naturally the sort of applications to manage; that is, each application component properties are studied piece by piece, understanding the lifecycle and dependencies of component, and then writing the state machine and causalities in CLOUDSCAPE.

Organisation. The remainder of this paper is organised as follows. Section 2 gives the intuition of the language through two real-world examples: Client-Server and Load Balancer. Section 3 discusses our syntax and operational semantics, illustrated by an example, and gives evidence of the effectiveness of the system through the reduction of the Client-Server example. Section 4 surveys related work and section 5 concludes with a discussion of possible future work for this system.

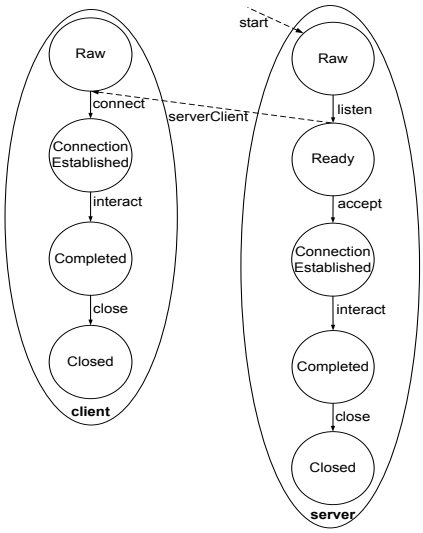

Figure 1. Lifecycle state diagrams of Client and Server components with the dependency `serverClient` between states Ready@Server and Raw@Client (in dashed arrow from source to target). The `start` dependency serves to initiate the computation of the server agent.

2. CLOUDSCAPE By Examples

This section gives an informal introduction to CLOUDSCAPE through a series of examples. Examples include coordination and control of components as in the Client-Server example, restoration of normal lifecycle in case of component failure as in the Client-Server example and, coordination and control of components added dynamically as in the Load Balancer example.

2.1 Client-Server Example

A server is created listening for connection from the client. Once a request for connection has arrived, the server accepts it establishing a connection with the client. The server interacts with the client and subsequently completes its run. Lastly, the server closes all the streams and sockets opened ahead. A client is created only connecting to a listening server. Once the connection is established, the client interacts with the server. The client ends by closing all the streams and sockets opened when the connection was established. The implementation in Java of the components, namely `Client` and `Server` (see bottom half of Figure 2), specifies all the methods that perform the fore-mentioned routines: create sockets, accept connection, interact with peer, and close connection. However, it does not express the constraint that the method (constructor) Client must take place after the method (constructor) Server has returned, where Client creates a client socket connected with the server and Server creates a server socket ready to accept connections from the client. For presentation reasons, we omit the full component's code[1], relegating it to a companion technical report [11].

Our solution to the above problem consists of two agents, namely `client` and `server`, controlling respectively the behaviour of each Java component along with the dependency that captures the constraint describing when to invoke the Client method: `serverClient` (see Figure 1). Each agent is described as a lifecycle state diagram, running transitions (`connect`; `interact`; `close` and `listen`; `accept`; `interact`; `close`) in the order given in the figure where at each state transition, the method of the component is invoked. The dependency `serverClient` ensures that the `connect` transition in the `client` takes place only after the `listen` transition in the `server` has occurred, consequently invoking the Client method after the Server method has returned. The `server` does not have any dependency on the `client`, therefore independently runs all transitions ensured by dependency `start`. Dependencies serve to describe constraints between lifecycle states of two agents and to define the runnability of an agent.

The implementation of the diagrams and dependencies in CLOUDSCAPE is given in the top half of Figure 2. A user-defined program is a list of dependencies and an attribute of name **Main** with expressions of objects and dependencies' instances. The names of objects, transitions and dependencies are the same as in the diagrams.

As mentioned in the introduction, an agent in CLOUDSCAPE is modeled as an object which is created by cloning another object, `Object` by default and updating attributes of the cloned object and adding new attributes. `server` and `client` objects are created cloning `Object` and, adding data attributes and behavior attributes or transitions as we refer to them in the paper. Data attributes are used to set up the components (`address` and `port`) and to store the state of component's lifecycle (`cState` and `sState`, initially set to *Raw*). Before presenting the intuition of transitions, we give the definition of *object*.

Definition 1 (CLOUDSCAPE object)**.** *An object consists of attributes that are defined over data fields and transitions. It describes the state diagram of a component lifecycle and models the computation entity that interprets the state diagram and controls the behaviour of a component. Computation happens mostly via predicate dispatch— an associated transition runs only when the guarding predicate is true.*

Transitions are defined as a block of statements guarded by a predicate on the component's data attributes. The `connect` and `listen` transitions in respectively `client` and `server` occur only if the state of the components is *Raw*. `connect` invokes the Client

[1] We use the same code as in the Java tutorial [6].

```
dependency start{true@src→trg}
dependency serverClient{src.sState≠"raw"@src→trg}

Main :
 let client = clone(Object)↔{
               address : "localhost";
               port : 1234;
               cState : "raw";


               connect : [cState="raw"]{
                 Client theClient = new Client(address, port);
                 cState : "connectionEstablished"
                };


               interact : [cState="connectionEstablished"]{
                 theClient.interact();
                 cState : "completed"
                };
               close : [cState="completed"]{
                 theClient.close();
                 cState : "closed"
                };
              },
 in
    start(unit, server);
    serverClient(server, client)
```

```
server = clone(Object)↔{
          port : 1234;
          sState : "raw";

          listen : [sState="raw"]{
              Server theServer = new Server(port);
              sState : "ready"
           };
          accept : [sState="ready"]{
              theServer.accept();
              sState : "connectionEstablished"
           };
          interact : [sState="connectionEstablished"]{
              theServer.interact();
              sState : "completed"
           };
          close : [sState="completed"]{
              theServer.close();
              sState : "closed"
           };
         }
```

```
class Client{
   Socket cSocket = null;
   ... // stream declarations

   public Client(String address, int port){
      try {
          cSocket = new Socket(address, port);
      }catch (UnknownHostException e) {
        ... // handle exception
      } catch (IOException e) {
        ... //handle exception
      }
   }
   ... /* definition of other methods: interact, close */
}
```

```
class Server{
   ServerSocket sSocket = null;
   ... // other socket and stream decl.

   public Server(int port){
       try {
            sSocket = new ServerSocket(port);
        } catch (IOException e){
            ... // handle exception
        }
   }
   ... /* definition of other methods: accept, interact,
          close */
}
```

Figure 2: Client-Server example: Modeling of the agents in CLOUDSCAPE (top) that control and coordinate the behaviour of the Java components (bottom).

constructor[2], and updates the state attribute with the new component's state *Connection Established*. In `server`, `listen` invokes the constructor Server, and updates the state attribute to *Ready*. In `client`, the `interact` transition controls the interactions with the server by invoking the `interact` method on the `Client` instance. Transitions may represent many steps in the computation of the system the object controls. `interact` starts the conversation only when the connection with the server is established and updates the state of the computation to *Completed* when the conversation has completed. `close` controls the end of the computation of the `Client` by invoking the method with the same name that closes all the streams and sockets opened ahead. In the server side, `accept` controls a ready `Server`, defined over a listening server socket, to accept a connection. `interact` and `close` control the computation of the `Server` similarly as in the `Client`. Below, we give the definition of *transition*.

Definition 2 (CLOUDSCAPE transition)**.** *A transition consists of a block of programming statements and a predicate. It provides a mechanism to control the behaviour of a component: the block of statements performs actions in an external language (Java in our study) and* CLOUDSCAPE*; the predicate guards the run of the block.*

The behaviour of `client` is guarded by the `serverClient` dependency (declared at the beginning of the program) at runtime, instantiated in the `let` scope. That is, `client` is active only if `server`'s attribute that stores the state of `Server` state(sState) is not *raw*. Hence, the behaviour that starts the `Client` will be invoked after the `Server` is listening for connections. The constraint described in the tutorial is expressed as a propositional expression on the lifecycle state of `Server`, guarding the behaviour of the agent that controls the `Client`. The behaviour of `server` is guarded by a dependency that does not enforce any constraint but

[2] Java code can be embedded in CLOUDSCAPE by injecting Groovy scripts. Groovy [24] is a scripting language that perfectly integrates with all features of Java and complements it with features from dynamic languages, including closures, maps, and regular expressions.

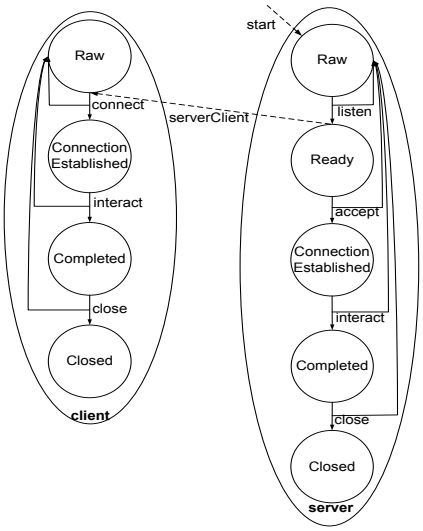

Figure 3. Lifecycle state diagrams of Client and Server restoring normal lifecycle in case of failure.

rather simply starts the behavior of the object, allowing transitions `listen`, `accept`, `interact` and `close` to be evaluated in the listing order. The `server` uses "unit" to denote lack of source object, in a similar sense as the "void" type in Java denotes lack of returning output. We use the term "unit" from ML. Below, we give the definition of dependency in CLOUDSCAPE.

Definition 3 (CLOUDSCAPE dependency)**.** *A dependency consists of a name, a boolean operator to compose instances of dependencies, a propositional expression to guard the transitions of an object and two input parameters to customize the propositional expression. The second parameter, denoted* `trg`*, represents the object that the propositional expression guards, called the* target object *and the first parameter, denoted* `src`*, represents the object that the propositional expression is defined, called the* source object*. Dependencies provide a mechanism to describe the constraints between the lifecycle states of two components and to define the runnability of agents that control components.*

2.2 Client-Server Example with Failure

Every method of each component in the Client-Server example may fail due to hardware or component logic, transiting the lifecycle state of a component to *Raw* as shown in the diagram of Figure 3. The Java runtime engine interrupts and exits abnormally the execution of the component in case of an erroneous action caused by component logic or hardware failure throwing an IOException or UnKnownHostException and so, affecting the normal lifecycle of the component. Consequently, the other JVM will interrupt the execution of the other component, throwing an IOException, due to the network failure caused by an abnormal close of the socket of the other peer. In the previous section, exceptions were handled at the component level using the Java exception handler, interrupting and exiting the running of one component, and consequently of the other, without notifying the agents. As a consequence, the lifecycle state machine represented in the agent is erroneously active on a state that does not match the real one.

We solve the problem of state consistency between agents and components in case of failure by shifting the exception handler of Java, that is provided through the `try − catch` clause, at the CLOUDSCAPE objects and augmenting the language with non-deterministic attribute-update. That is, exceptions must be handled

```
client = clone(Object)↔{
            address : "localhost";
            port : 1234;
            cState : "raw";

            connect : [cState="raw"]{
                try{
                    Client theClient = new
                      Client(address, port);
                    cState : "conEst"
                }catch(UnknownHostException e){
                    ... /* handle exception for
                           component */
                    cState : "raw"
                }catch(IOException){
                    ...
                    cState : "raw"
                }
            };
            ... /* definition of other transitions:
                   interact, close */
          }

class Client{
   Socket cSocket = null;
   ... // stream declarations

   public Client(String address, int port) throws
                 IOException, UnKnownHostException{
       cSocket = new Socket(address, port);
   }
   ... /* definition of other methods: interact, close */
}
```

Figure 4: Client-Server example with failure: Modeling of the agent in CLOUDSCAPE (top), controlling and restoring the normal lifecycle of Client component (bottom) in case of failure by shifting the Java exception handler at the definition of agent.

at CLOUDSCAPE objects to provide a sound model that controls and coordinates components in presence of failure. In the Client-Server example (see Figure 4), if an exception is raised in one of the methods `connect`, `listen`, `accept`, `interact`, and `close`, the transitions of the same name, that control those methods, restore the computation of the component to the initial state[3]: the state (attribute) update is defined inside the Java clause `catch`, otherwise the transitions follows the normal lifecycle: the state update is defined after the Java code. In the formal model, we define the two attribute updates composed in parallel through a non-deterministic operator as we shall see later. As mentioned above, the exception will propagate to the other component, resulting in the two agents re-initiating computation from the initial state (*Raw*) and consequently, creating new instances of `Client` and `Server`.

2.3 Load Balancer Example

A load balancer is created to manage and dispatch the load of work to several web servers in relation to time response. It creates a new web server if the response time of a service is greater than a certain threshold— the reasonable response time. The work is dispatched to the new web server once it is created. A web server is created to accept requests of work from a load balancer and to process them. For presentation reasons, we have simplified the specification of the problem to only the dynamic features of it. The implementation in Java of the components, namely Load Balancer

[3] Restoring the normal lifecycle to the initial state is related to the particular logic of this example and should not be considered as a pattern on how to restore normal lifecycle in case of failure in CLOUDSCAPE.

```
dependency start{true@src → trg}
dependency webServerLB { &(src.wsState="created")@src → trg}

Main :
 let LB = clone(Object)↔{                                    WS = clone(Object)↔{
             wsInstncs : 0;                                          wsState : "raw";
             threshold, respTime : 1000;
             lbState : "raw";

             create : [lbState="raw"]{                               create : [wsState="raw"]{
                       new LoadBalancer(respTime, wsInstncs);                  WebServer ws = new WebServer();
                       lbState : "ready"                               wsState : "ready"
             };                                                      };
             createWS : [respTime>threshold & lbState="ready"] {     accept : [wsState="ready"]{
                       let ws=clone(WS)                                        ws.accept();
                       in{                                             wsState : "perform"
                          start(unit, ws);                             };
                          webServerLB(ws, LB);
                          lbState : "newComp"
                       }
             };
             increaseWSInstncs : [lsState="newComp"]{                run : [wsState="perform"]{
                         wsInstncs : wsInstncs+1;                              ws.run();
                         lbState : "ready"                             wsState : "ready"
             }                                                       }
     }                                                           }
in
   start(unit, LB)
```

Figure 6: Load Balancer example: Modeling of the load balancer and web server agents in CLOUDSCAPE.

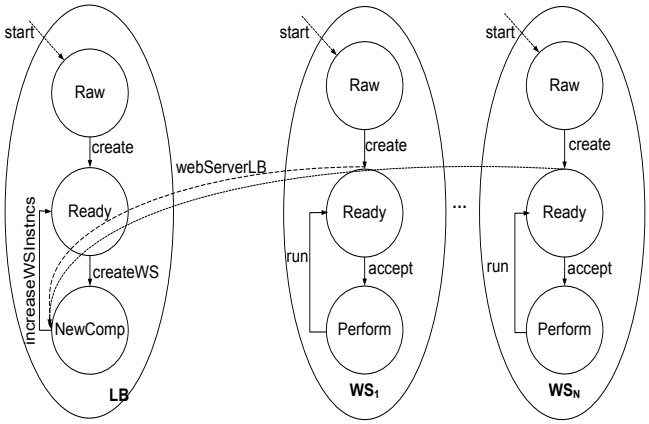

Figure 5. Lifecycle state diagrams of Load Balancer and Web Server(s) with the dependency `webServerLB` between states Created@WebServer and NewComp@LoadBalancer. The `start` serves to initiate the computation of the load balancer and web server(s) agent as in the Client-Server example.

and Web Server (see [15]), specifies all the methods that perform the fore-mentioned routines: define variables that store the number of Web Servers and service response time, set up a web server, accept requests of work, process work loads. However, it does not express the constraint that the value of the variable that stores the number of Web Servers must be increased after the new Web Server is created.

Our solution to the above problem consists of two prototype agents, namely `LB` and `WS`, controlling respectively the behaviour of each Java component along with the dependency that captures the constraint describing when to increase the value of the variable storing the number of Web Server instances (see Figure 5). The `LB` agent creates instances of `WS` agents to maintain a reasonable time response and so, updates the value of the variable that stores the number of Web Server instances. That variable is then used by the Load Balancer component to dispatch the load of work. Instances of the `webServerLB` dependency are created dynamically from `LB` to ensure that the `increaseWSInstncs` transition takes place only after the `create` transition in `WS` has occurred, consequently increasing the value of the variable storing the number of Web Server instances after the new instance of Web Server is created. The `WS` instances do not have any dependency on the `LB`, therefore independently run all transitions through the dependency `start`.

Figure 6 gives the modeling of the diagrams and dependencies in CLOUDSCAPE. `LB` and `WS` contain attributes that store the state of components lifecycle, e.g. `lbState` and `wsState`, initially set to *Raw*. The `respTime` attribute stores the response time of a service and is updated in the Load Balancer component code; `threshold` stores the reasonable response time; `wsInstncs` stores the number of web server created and its value is used by the Load Balancer component code when dispatching the workload.

In `LB` and `WS`, the `create` transition controls the creation of respectively an instance of the Load Balancer and Web Server classes, and updates the state attribute with the new components state *Ready*. The constructor of Load Balancer sets up an instance (with the values of number of Web Server instances and response time) that dispatches the work load in base of the number of Web Server. The constructor of Web Server creates an instance ready to accept loads of work from Load Balancer. In `LB`, the `createWS` transition controls the creation of new `WS` objects and instances of `webServer` dependency, when the response time of a service is lower than a certain threshold (The dependency is added dynamically to the `LB` global predicate after `WS` is created in the memory). The `increaseWSInstncs` transition increases the number of Web Server instances stored in `wsInstncs`. In `WS`, `accept` controls a ready Web Server to accept a load of work, and `run` processes the load of work by invoking respectively the methods with the same name.

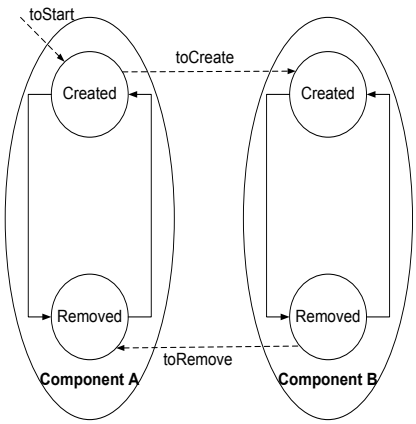

Figure 7. Lifecycle state diagram of Two Components example.

During the evaluation of the `createWS` transition, the propositional expression in the dependency `webServerLB` composes with the propositional expression of the `LB`'s predicate through the & operator; thus, making `LB` runnable only when both the proposition in the dependency and the predicate evaluate `true`. The `LB`'s predicate is defined by the composition of the propositional expression in the `start` and instances of `webServerLB` (created for each web server in the session) dependencies. By specifying the boolean operator inside the definition of dependency, our model provides a simple mechanism to add dependencies dynamically as the `webServerLB` dependency. That is, this design gives a simpler definition of our language than the one that specifies the boolean operators as constructs of the grammar of dependency instance. The `start` dependency is used in the same way as in the Client-Server example: to start both agents.

3. Formal Model

We now introduce the core CLOUDSCAPE language to formalize the intuitions given above. This section contains the syntax and operational semantics. For maintaining a balance between the intuitions and mathematical definitions given in this paper, we relegate some of the operational semantics rules in a companion technical report [11]. The operational semantics rule are illustrated through a system of two components. Evidence of the validation of our design and its effectiveness is given by the reduction of the Client-Server management system.

3.1 Two Components Example

Consider a system of two components where the logic of each component, namely `A` and `B`, consists of creating and removing an entity, e.g. a virtual machine (This is a typical scenario for management systems of *IaaS* where virtual machines are offered as a service to customers). The specification of the system defines component `A` creating first an entity and then component `B` creating a second entity. `B` can create and remove an entity repetitively in the system until `A` has removed its entity. The last action takes place only when `B` has removed its entity.

The diagram in Figure 7 shows the lifecycles state diagrams of the two component system. Each component includes two states, namely `Created` and `Removed`. The `toStart` dependency starts the session, while dependencies `toCreate` and `toRemove` define respectively the constraint on the order of creating and removing entities between components `A` and `B`.

3.2 Syntax

Figure 8 provides the syntax of our language. The metavariable D stands for dependencies; P stands for propositional expression; e (as well as with suffix and g) stands for expressions; v stands for values. The metavariable N ranges over dependency names; `src`, `trg` range over dependency variables; x, y, ... range over variables; k, l, ... range over attributes names; L, L', L'', ..., including Object and Main, range over locations; n ranges over naturals.

The construct `dependency` N{&/|/ P@x→ y} describes a dependency between two objects, placehold by x and y, as a propositional expression P. Dependencies are composed by the & and | operators to model multiple dependencies. That is, instances of different or same dependency can be composed using the fore-mentioned operators. The definition of the operator inside the definition of dependency provides a simple mechanism to add dependencies dynamically as explained in Section 2.3.

Propositional expressions include boolean values, data attributes over boolean values x.k and L.k (L.k is part of the runtime syntax— syntax of the language introduced at runtime), in- equality tests ($<$, $>$, =) on data attributes, two propositional expressions composed using the boolean operators & and |, and a propositional expression prefixed by !. The operators read the same as in Java.

Dependencies of the Two Components example are defined as follows:

$$\textbf{dependency}\ toStart\{\ !trg.created@src\rightarrow trg\}$$
$$\textbf{dependency}\ toCreate\{src.created@src\rightarrow trg\}$$
$$\textbf{dependency}\ toRemove\{\ |\ (src.removed)@src\rightarrow trg\}$$

The $toStart$ dependency captures the attribute `created` of the target object as `false`; $toCreate$ captures the attribute `created` of the source object as `true`; $toRemove$ captures the attribute `removed` of the source object as `true`. The propositional expression of the latter is prefixed by the | operator to compose it with the propositional expression of $toStart$ when guarding the behavior of the object that controls component `A`. The boolean operators are left-associative, when grouping more than two propositional expressions.

Expressions define behaviour in CLOUDSCAPE, including attribute names k, values v and richer expressions using constructs: *attribute*, *cloning*, *application of dependency*, *non-deterministic update*, *sequential* and *parallel composition*.

The attribute construct $e : e'$ describes an attribute of name e and expression e'. For attribute names k, e' must denote values v and for locations L, including Main, e' must denote expressions, including transitions (These constraints are ensured by the operational semantics rules in Figure 9). Attribute of the form $L : e$ denotes the expression of each object (location L in the store) in the evaluation context where L is similar as the process ID (pid) in the context of operating systems. This form is used to define a spawning policy of new objects created during the evaluation of expression e as we shall see in the operational semantics rule.

The cloning construct `clone` (e) ↤ {e'} describes the creation of an object by cloning another object e, Object by default, and update it with new attributes e' similarly as in the system of Fisher *et al.* [16]. e' must denote attributes of the form k:v (This constraint is ensured by the operational semantics rules in Figure 9).

The application construct N(e, e') describes the application of two objects e and e' to the dependency of name N. e and e' must denote locations or attribute names (This constraint is ensured by the operational semantics rules in Figure 9).

The non-deterministic construct $e \oplus e'$ describes an (data) attribute update by one of two values to represent a normal transition of one state to another following component's logic and an exception transition to restore normal lifecycle due to component failure.

D	::=	dependency N{&/|/ [4] P@x→ y}
P	::=	true | false | x.k | L.k | Pop v | P& P' | P | P' | !P
e	::=	k | v | e: e' | clone (e) ↔ {e'} | N(e, e') | e⊕e' | e;e' | e|e' | **0**
v	::=	object/ [5] [P]{e} | L | n | true | false ⋯
H	::=	$L \mapsto Odescr$, H | Object ↦ []
$Odescr$	::=	$Odescr$↔ k : v | ∅
Env	::=	N ↦ &/|/ P@x→ y, Env | ∅

Figure 8. User and run-time syntax.

e and e' must denote (data) attribute-update (This constraint is ensured by the operational semantics rules in Figure 9).

The sequential composition and parallel composition are standard. **0** signifies the end of an object behavior. Parallel composition and **0** are part of the runtime syntax. Values include transitions and primitive values such as natural numbers and boolean values.

Transitions labeled object represent the guarded behavior of an object, modeling at runtime the behavior of the agent, where instances of dependencies upon the (target) object define the predicate (guard) and the transitions (behaviour) associated to the object define the scope of the expression (block of statements). We will refer throughout the paper to the guarded behaviour of an object as *object transition* and to the single transition associated to an object as *transition*. The predicate of object transitions is typically defined over other objects attributes (global), supporting a "read-anything" capability, while in transitions, the predicate and attribute update is defined over object's own data attributes, supporting a "read/write-owner" capability as in UNIX (These constraints are ensured by the operational semantics rules in Figure 9).

In the remaining part of the Two Component program, the attribute Main with the prototype object *O* and its instances, namely objects A and B, and instances of dependencies looks like this:

```
Main:
 let O = clone(Object) ↔ {created, removed : false;
                          create : [!created]{
                                     #create entity
                                     created : true;
                                     removed : false};
                          remove : [created & !removed]{
                                     #remove entity
                                     removed : true;
                                     created : false}
                          }
 in
   let A = clone(O) ↔ {name:A}, B = clone(O) ↔ {name:B}
   in
      toStart(unit, A);toCreate(A, B);toRemove(B, A)
```

where the dependency $toStart$ is customized on the initial state of A and is used to start the computation of A; the dependency $toCreate$ is customized on the state of *A* as "created", guarding the behaviour of B; the $toRemove$ dependency is customized on the state of B as "removed", guarding the behaviour A with $toStart$. Each agent modeled by objects *A* and *B* controls respectively components `A` and `B`. In the *create* and *remove* transitions, the line starting with # denotes the invocation of components respectively to create and remove an entity.

[4] &/|/ denotes either & or | or none of them.

[5] object/ denotes either object or none.

Objects attributes are kept in a store that is a pair of object location and description. The object description is a sequence of value attributes. The base case for the store is the pair Object and empty list of value attributes, and the base case for object description is the empty list. Our attribute-based object encoding is similar to standard object encodings [9, 16]. Dependencies are kept in an environment Env— a pair of dependency name and propositional expression defined over two variables possibly prefixed by a boolean operator.

Encoding of the* let *construct In Section 2, we used the let construct to define more easy to read and understand programs. The let variable binding construct can be simulated in our language, using the attribute and sequence constructs as shown below.

$$let\ \mathrm{k} = e\ in\ e' \triangleq \mathrm{k} : e; e'$$

3.3 Operational Semantics

Figure 9 gives the operational semantics in a small step style via the reduction relation ⟶ where the state of the machine is defined only by terms of the calculus and store, written "$H; e \longrightarrow H'; e'$", read "the configuration $H; e$ of expression e and store H reduces to the configuration $H'; e'$ of expression e' and store H' in one step". Sometime, we will refer to e as the *evaluation context*. The interesting features of the rules are how they create a new object in the store and in the evaluation context, add a dependency to an object transition, spawn a behavior, evaluate a (object) transition, update (non-deterministically) an attribute, and parallel evaluation of two behaviors preserving a linear consistent shared memory.

Creation of new objects. New objects are created in the store through rule *R-Clone*, storing the attributes of the cloned object and those added by the user. The attributes of the cloned object are replaced by the user one if they have the same name, using the ⊎ operator defined in Figure 10. The actions of reading the store are atomic where the store contains the records of the cloned object (L'). The new location created is substituted into the occurrences of the reference (k′) in the remaining expression.

The three objects of the Two Components example are created like this:

H; **Main**: P $\longrightarrow^{[R-Clone](3)}$ [6]

H'; **Main**: $toStart(unit, L'); toCreate(L', L'');$
$toRemove(L'', L') \longrightarrow^{[R-New](2)}$

where P denotes the program (objects and dependency instances) and H denotes the store containing only the pair Object and empty list. For presentation reasons, we omit the line starting with # from the definition of objects in the memory and evaluation context. The new store containing the objects is defined below:

Object	↦	[]
L	↦	$[created:$ false, $removed:$ false, $create:$ $[!created]\{created$: true; $removed:$ false}, $remove:$ $[created\&!removed]\{removed:$ true; $created:$ false}]
L', L''	↦	$H(L) \uplus \{name : A\}, H(L) \uplus \{name : B\}$

The behaviour of a new object (object transition) is created in the evaluation context only when the first dependency is applied to it through rule *R-New*. This design allows prototype objects — objects that describe the behavior of runnable objects—- to not be part of the evaluation context. The behavior of the target object is

[6] The number in parenthesis associated to a rule name denotes the number of time that rule is applied.

$$\frac{H(L') = Odescr \quad L'' \notin dom(H) \quad H' = H[L'' \mapsto Odescr \uplus \overline{\mathtt{k} : v}]}{H; L : \ \mathtt{k'}{:}\mathsf{clone}(L') \leftrightarrow \{\overline{\mathtt{k} : v}\}; e' \longrightarrow H'; L : e'\{L''/\mathtt{k'}\}} \ L' \in dom(H) \quad R - Clone$$

$$\frac{P = Env(N)\{L'/\mathtt{x}\}\{L''/\mathtt{y}\} \quad \mathtt{transitions}(H(L'')) = \overline{[P']\{e'\}}}{H; L : N(L', L''); e \longrightarrow H; L : (e|L'' : \mathsf{object}\ [P]\{\overline{[P']\{e'\}}\})} \ L'' \in dom(H), L'' \notin dom(e) \ R - New$$

$$\frac{\&/|\ P' = Env(N)\{L'/\mathtt{x}\}\{L''/\mathtt{y}\}}{H; L : (N(L', L''); e|L'' : \mathsf{object}\ [P]\{e'\}) \longrightarrow H; L : (e|L'' : \mathsf{object}\ [P\ \&/|\ P']\{e'\})} \ R - DepO$$

$$H; Main : (L_1{:}\mathsf{object}\ [P_1]\{e_1\} \mid ... \mid L_n{:}\mathsf{object}\ [P_n]\{e_n\}) \longrightarrow H; L_1{:}\mathsf{object}\ [P_1]\{e_1\} \mid ... \mid L_n{:}\mathsf{object}\ [P_n]\{e_n\}\ \textit{R–SpawnM}$$

$$\frac{\mathtt{eval}(H, P) = \mathsf{true}}{H; L : \mathsf{object}\ [P]\{e\} \longrightarrow H; L : e; \mathsf{object}\ [P]\{e\}} \ \mathtt{Loc}(P) \subseteq dom(H) \quad R - ObjectT$$

$$\frac{\forall j \in \{1..l-1\}.\mathtt{eval}(H(L), P_j) = \mathsf{false} \quad \mathtt{eval}(H(L), P_l) = \mathsf{true}}{H; L : \overline{[P_i]\{e_i\}}_{1..n}; \mathsf{object}\ [P]\{e\} \longrightarrow H; L : e_l; \mathsf{object}\ [P]\{e\}} \ (l \in \{1..n\}), L \in dom(H) \quad R - TranT$$

$$\frac{\forall j \in \{1..n\}.\mathtt{eval}(H(L), P_j) = \mathsf{false}}{H; L : \overline{[P_i]\{e_i\}}_{1..n}; \mathsf{object}\ [P]\{e\} \longrightarrow H; \mathbf{0}} \ L \in dom(H) \quad R - TranF$$

$$\frac{\mathtt{k} \in dom(H(L)) \quad H' = H(L)[\mathtt{k} \rightarrow \mathtt{v}]}{H; L : \mathtt{k}{:}e; e' \longrightarrow H'; L : e'} \ (e \downarrow v), L \in dom(H) \quad R - Attribute$$

$$\frac{H_1; e \longrightarrow H_1'; e' \quad H_2; g \longrightarrow H_2'; g' \quad dom(H_{1\alpha}') \cap dom(H_2') = \emptyset}{H_1, H_2; e|g \longrightarrow H_{1\alpha}', H_2'; e_\alpha'|g'} \ dom(H_1) \cap dom(H_2) = \emptyset \quad R - Par$$

$$\frac{H; e \longrightarrow H'; e'}{H; e|g \longrightarrow H'; e'|g} \ R - IPar$$

Figure 9. Operational Semantics

created inside the context of the evaluating object; thus, prohibiting it to run until the running transition has terminated. This design allows to capture all the dependencies instances on the target object that may be present in the running transition before spawning the behaviour of the object (As mentioned in Section 3.2, we use the attribute construct $L : e$ to express the evaluation context of an object). The object transition contains the propositional expression of the dependency as predicate and the associated transitions of the target object as body. `transitions` looks up an object descriptor for transitions (see Figure 10).

Objects behavior of the Two Component example are created by respectively dependency $toStart$ and $toCreate$:

$$H'; \textbf{Main}: (toRemove(L'', L') \mid L'{:}\mathsf{object}[!L'.created]\{create{:}...; remove{:}...\} \mid L''{:}\mathsf{object}[L'.created]\{create : \ ...; remove : \ ...\}) \longrightarrow^{[R-DepO]}$$

Object O is a prototype object while A and B are runnable objects. The behavior of A and B (referenced respectively by L' and L'') are placed in the evaluation context of the main program, disallowing it to run until the other dependency ($toRemove$) is added.

Adding a dependency. Dependencies are added to guard the behavior of an object (yet not runnable) through rule *R-DepO*. `Env` defines the environment of dependencies for a program. The proposition returned is instantiated by replacing `x` with the first argument and `y` with the second argument. Subsequently, it is added to the guard of the second object's behaviour, following the composition rules of | or &.

The $toRemove$ dependency in the example is added to the guarding predicate of L' like this:

$$H'; \textbf{Main}: (L'{:}\mathsf{object}[!L'.created \mid L''.removed]\{create : \ ...; remove : \ ...\} \mid L''{:}\mathsf{object}[L'.created]\{create : \ ...; remove : \ ...\}) \longrightarrow^{[R-SpawnM]}$$

where the propositional expression is composed with the one of the predicate through the | operator present in the $toRemove$ dependency.

Spawning behaviors. Rule *R-SpawnM* spawns the behavior of new objects when the main program is completely evaluated. Another rule (see [11]) defines this action during the evaluation of a transition.

In the example, the two behaviors are spawned to run independently in parallel, resulting in:

$$H'; L'{:}\mathsf{object}[!L'.created \mid L''.removed]\{create : ...; remove : ...\} \mid L''{:}\mathsf{object}[L'.created]\{create : \ ...; remove : \ ...\} \longrightarrow^{[R-IPar, R-ObjectT]}$$

Evaluation of transitions. The rule *R-ObjectT* reduces the object transition to the transitions associated to the object, appending the object transition at the end to provide continuity of computation, if the predicate evaluates true (see definition of `eval` for store H in

Union of attributes

$$\{..., \mathrm{k}\ v, ...\} \uplus \{\mathrm{k}\ v'\} = \{..., \mathrm{k}\ v', ...\}$$
$$\{..., \mathrm{k}\ v, ...\} \uplus \{\mathrm{k}'\ v'\} = \{..., \mathrm{k}\ v, ..., \mathrm{k}'\ v'\}$$
$$\{..., \mathrm{k}\ v, ...\} \uplus \{\mathrm{k}'\ v', \overline{\mathrm{l}\ g}\} = \{..., \mathrm{k}\ v, ...\} \uplus \{\mathrm{k}'\ v'\} \uplus \{\overline{\mathrm{l}\ g}\}$$

Transitions look up

$$\texttt{transitions}(Odescr \leftrightarrow \mathrm{k}\ v) = \texttt{transitions}(Odescr) \cup \texttt{transitions}(v)$$
$$\texttt{transitions}([P]\{e\}) = \{[\mathrm{P}]\{\mathrm{e}\}\}$$

Evaluation of predicates

$$\texttt{eval}(H/Odescr, \mathsf{true}) = \mathsf{true} \qquad \texttt{eval}(H/Odescr, \mathsf{false}) = \mathsf{false}$$
$$\texttt{eval}(H, L.\mathrm{k}) = v_i \text{ if } H(L) = [\mathrm{k}_1 v_1, ..., \mathrm{k}_n v_n] \text{ and } \mathrm{k}_i = \mathrm{k} \text{ where } i \in \{1..n\}$$
$$\texttt{eval}([\mathrm{k}_1 v_1, ..., \mathrm{k}_n v_n],\mathrm{k}) = v_i \text{ if } \mathrm{k}_i = \mathrm{k} \text{ where } i \in \{1..n\}$$
$$\texttt{eval}(H/Odescr, P\ \mathsf{op}\ v) = \texttt{eval}(H/Odescr, P)\ \mathsf{op}\ v$$
$$\texttt{eval}(H/Odescr, P\&/|P') = \texttt{eval}(H/Odescr, P)\&/|\texttt{eval}(H/Odescr, P')$$
$$\texttt{eval}(H/Odescr, !P) = !\texttt{eval}(H/Odescr, P)$$

Location look up

$$\texttt{Loc}(\mathsf{true}) = \emptyset \qquad \texttt{Loc}(\mathsf{false}) = \emptyset \qquad \texttt{Loc}(L.x) = \{L\}$$
$$\texttt{Loc}(P\ \mathsf{op}\ v) = \texttt{Loc}(P) \qquad \texttt{Loc}(P\&/|P') = \texttt{Loc}(P) \cup \texttt{Loc}(P') \quad \texttt{Loc}(!P) = \texttt{Loc}(P)$$

Figure 10. Auxiliary definitions

Figure 10); otherwise, it reduces to itself through rule *R-ObjectF* (see [11]). The store must contain the locations of objects present in the predicate of the object transition (see definition of Loc in Figure 10).

The two behaviors of the example reference the same memory location in the global predicate (see L' in the resulting expression of rule *R-SpawnM*), allowing only one object transition to evaluate in one step as we shall see in the definition of rule *R-IPar*. We present the most interesting evaluation path in the example via rule *R-ObjectT* on L'; the other object transition evaluates to itself through rule *R-ObjectF*.

$$H'; L'{:}create : ...; remove : ...;$$
$$\quad \mathsf{object}[!L'.created \mid L''.removed]\{create : ...; remove : ...\}$$
$$|L''{:}\mathsf{object}[L'.created]\{create : ...;$$
$$remove : ...\} \longrightarrow^{[R-IPar, R-TranT]}$$

The rule *R-TranT* looks up in the list of transitions associated to the object for a transition that predicate evaluates to true and so, returns its block of statements. Rule *R-TranF* signifies the end of an object's behavior since no transition is available to run; i.e. the predicate of all transitions evaluate to false. In contrast to the rules for evaluating object transitions, the rules for transitions allow predicates to be defined strictly over object's attributes by restricting the scope of the store H to $H(L)$ when evaluating the predicates (see definition of eval for object descriptor $Odescr$ in Figure 10). A clear separation at the language definition between the kind of attributes used in dependencies and predicates of transitions educates programmers to use the state machine metaphor when modeling lifecycle management of distributed applications, where transitions control the change of lifecycle state of a component and dependencies express the causalities between the lifecycle states of two components.

In the Two Component example, L' evaluates the first transition through rule *R-TranT*, resulting in:

$$H; L' : created : \mathsf{true}; removed : \mathsf{false};$$
$$\quad \mathsf{object}[!L'.created \mid L''.removed]\{create : ...; remove : ...\}$$
$$|L''{:}\mathsf{object}[L'.created]\{create : ...;$$
$$remove : ...\} \longrightarrow^{[R-IPar, R-Attribute]}$$

Update an attribute. The value of an object's attribute can be updated from the scope of a local transition associated to the object through rule *R-Attribute*. The expression $(e \downarrow v)$ evaluates the expression e to the value v.

In the example, L' updates the attribute *created*, resulting in:

$$H''; L' : removed : \mathsf{false};$$
$$\quad \mathsf{object}[!L'.created \mid L''.removed]\{create : ...; remove : ...\}$$
$$|L''{:}\mathsf{object}[L'.created]\{create : ...;$$
$$remove : ...\} \longrightarrow^{[R-IPar, R-ObjectT]}$$

where the new store reflects the update of L' as:

$$L' \quad \mapsto \quad H'(L') \uplus \{created : true\}$$

The remaining reduction steps follow a similar usage of rules as described above so we leave them to the curious reader.

An object can non-deterministically update its data attribute. Below, we present the left update where the name of the attribute can be the same in both sides of the construct. Symmetrically is defined the right update.

$$H; L : \mathrm{k}{:}e \oplus \mathrm{k}'{:}e'; g \longrightarrow H; L : \mathrm{k}{:}e; g$$
$$H; L : \mathrm{k}{:}e \oplus \mathrm{k}'{:}e'; g \longrightarrow H; L : \mathrm{k}'{:}e'; g$$

The computation follows on the data-attribute update chosen.

Parallel evaluation. The rule *R-Par* evaluates two programs in parallel when the locations of one's memory are different from the other to maintain an atomic (linear) consistent shared memory. The α operation applied to the new store renames the locations that may be created during the evaluation of the first program, in a similar way as the α-conversion in the lambda calculus renames the names of variables in a lambda expression. This design avoids clashes of names when the two stores are composed sequentially; the α operation is also applied to the expression to maintain a coherence between the location names in the evaluation context and store.

This rule defines also how two object transitions evaluate in parallel, partitioning the memory to satisfy the side conditions present in the computation rules and so, maintaining a linear shared memory; e.g. an object transition and an attribute-update rule can take place in parallel if the latter does not affect the state of any of the objects present in the predicate of the object transition.

If the memory can not be partitioned to satisfy the memory conditions in the rules, then the two object can always interleave the evaluation with each other non-deterministically following rule *R-IPar*.

Other rules add dynamically a dependency to the running object transition after it has been evaluated.

Properties of the operational semantics. We prove that our operational semantics maintains a linear consistent shared memory.

For the proof, we define a function that returns the list of locations read and written during a one step evaluation of an expression as $H; e \longrightarrow H'; e' \Rightarrow \overline{L}$. The full definition and proofs are given in a companion technical report [11].

Lemma 3.1. *If* $H; e \longrightarrow H'; e' \Rightarrow \overline{L}$ *then* $\{\overline{L}\} \subseteq dom(H)$.

Theorem 3.2 (Linear consistent memory)**.** *If* $H; e \longrightarrow H'; e' \Rightarrow \overline{L}$ *then* $\overline{L}$ *has no duplicates.*

Proof. By induction over the operational semantics rules and Lemma 3.1. □

3.4 Reducing Client-Server

We give evidence of the effectiveness of the formal model and validate its design through the reduction of the Client-Server example described in Section 2.1. The reduction steps show how the agents modeled by objects `client` and `server` control and coordinate the execution flow between the Java components, creating the client socket only after the server socket is listening for connections. The initial store contains only the Object location as in the Two Component example. In the reduction steps below, P and Q denote respectively the clone expressions for the client and sever object in Figure 2.

$$H; \textbf{Main}: client : P; server : Q; start(unit, server); serverClient(server, client) \longrightarrow^{[R-Clone]}$$

$$H'; \textbf{Main}: server : Q; start(unit, server); serverClient(server, L) \longrightarrow^{[R-Clone]}$$

$$H''; \textbf{Main}: start(unit, L'); serverClient(L', L) \longrightarrow^{[R-New]}$$

where H'' contains records for locations L and L', defined below:

$$\begin{array}{lll} \text{Object} & \mapsto & [] \\ L & \mapsto & [address : \text{“}localhost\text{”}, port : 1234, \\ & & cState : \text{“}raw\text{”}, connect : [cState = \text{“}raw\text{”}]\{...\}, \\ & & interact : [cState = \text{“}connectionEst\text{”}]\{...\}, \\ & & close : [cState = \text{“}completed\text{”}]\{...\}] \\ L' & \mapsto & [port : 1234, sState : \text{“}raw\text{”}, \\ & & listen : [sState = \text{“}raw\text{”}]\{...\}, \\ & & accept : [sState = \text{“}ready\text{”}]\{...\}, \\ & & interact : [cState = \text{“}connectionEst\text{”}]\{...\}, \\ & & close : [cState = \text{“}completed\text{”}]\{...\}] \end{array}$$

$$H''; \textbf{Main}: (serverClient(L', L) \mid L' : Q') \longrightarrow^{[R-New]}$$

where $Q' \triangleq$ object [true]{
$listen : [sState = \text{“}raw\text{”}]\{...\}$,
$accept : [sState = \text{“}ready\text{”}]\{...\}$,
$interact : [sState = \text{“}connectionEst\text{”}]\{...\}$,
$close : [sState = \text{“}completed\text{”}]\{...\}$
}

$$H''; \textbf{Main}: (L' : Q' \mid L : P') \longrightarrow^{[R-SpawnM]}$$

where $P' \triangleq$ object $[L'.sState \neq \text{“}raw\text{”}]$\{
$connect : [cState = \text{“}raw\text{”}]\{...\}$,
$interact : [cState = \text{“}connectionEst\text{”}]\{...\}$,
$close : [cState = \text{“}completed\text{”}]\{...\}$
}

$$H''; L' : Q' \mid L : P' \longrightarrow^{[R-Par, R-ObjectT, R-ObjectF]}$$

Only the predicate of `server` (L') object transition evaluates true under an empty store.

$$\emptyset, H_2''; L' : Q''; Q' \mid L : P' \longrightarrow^{[R-IPar, R-TranT]}$$

where $Q'' \triangleq listen$: $[sState = \text{“}raw\text{”}]\{...\}, accept$: $[sState = \text{“}ready\text{”}]\{...\}, interact$: $[sState = \text{“}connectionEst\text{”}]\{...\}, close$: $[sState = \text{“}completed\text{”}]\{...\}$ and $H_2'' = H''$.

$$H''; L' : Server\ theServer = new\ Server(port); sState : \text{“}ready\text{”}; Q' \mid L : P' \longrightarrow^{[R-IPar, Java]}$$

$$H''; L' : sState : \text{“}ready\text{”}; Q' \mid L : P' \longrightarrow^{[R-IPar, R-Attribute]}$$

$$H'''; L' : Q' \mid L : P' \longrightarrow^{[R-Par, R-ObjectT, R-ObjectT]}$$

where $H''' = H''(L') \uplus \{sState : \text{“}ready\text{”}\}$.

$$\emptyset, H_2'''; L' : Q''; Q' \mid L : P''; P' \longrightarrow^{[R-Par, R-ObjectT, R-ObjectT]}$$

Thus, the `Client` component starts computation when the `Server` component has created a listening socket, following the logic of creating a socket connection. The remaining steps follow the same rules as presented to this step, so we leave them to the curios reader.

4. Related Work

SmartFrog. The idea of dependency modeling in CLOUDSCAPE originates from previous works by the authors on management of federated systems [17] in SmartFrog (SF). The initial work provides simply a general idea on how to use dependency modeling to manage highly distributed, federated entities as an alternative to workflow approaches. SF [8] is a language used mostly for modeling the deployment of components on multiple hosts. In addition, it provides a Java library used to read and write the attributes of SF objects from components code. SF memory model of objects is designed following the *blackboard* metaphor [12] — a shared space in which a problem is decomposed and incrementally solved. An immediate advantage of the blackboard approach is extendibility, new components can be added into a system without changing the data flow of the system. The blackboard metaphor consists of an arbiter that decides which object to run in the case when more than one object is active. In contrast, CLOUDSCAPE semantics allows objects to run independently and communicate with each other through dependencies. Despite its maturity, SF does not support coordination and control of components in a distributed application and so, leaving unsolved the two problems of this paper. However, SF offers an interesting platform to implement and further develop CLOUDSCAPE.

Agent-oriented abstractions. CLOUDSCAPE approach is similar to the A&A(Agents and Artifacts) meta-model [21, 22] used to model multi-agents systems. In the meta-model, agents model the logic and control of components' activities and artifacts model function-oriented components. However, the meta-model does not provide any support to express dependencies between the activities of two agents, restore the normal computation in case an activity fails and add activities dynamically. `simpA` [23] is a Java library that uses the A&A meta-model to provide support for designing the architecture of multi-threaded/concurrent applications. Agents in `simpA` are classes with methods extended by notations to specify the logic and control of activities that define the concurrent application. While, in CLOUDSCAPE, agents are objects extended with transitions to (1) specify the control of activities of a component and (2) express dependencies between transitions of other agents.

Workflow approach. The current state-of-the-art in tools for service automation and lifecycle management (for the Cloud) include HP Server Automation and Operations Orchestration [5], ControlTier [3] and Capistrano [1], which provide dashboard-driven workflow-based management of services, and node configuration tools like Chef [2] and Puppet [7]. The use of workflow to manage service deployments in the Cloud has a number of shortcomings. It is inherently not scalable, hard to maintain, and does

not promote reuse. Instead of managing scripts for every eventually in managing service artifacts, we push the control logic down to the management components themselves. In this way, CLOUDSCAPE addresses the issue of coordination between tasks, following the structured, distributed approach to design more robust and scalable management systems.

Other languages and systems. Other frameworks have been developed to model distributed computation in the Cloud, namely Hadoop [4], MapReduce [13], Dryad [18] and Skywriting [20]. An aspect that makes these frameworks successful to exploit the hardware on data centers when compared to mainstream programming languages is the high-level API on sockets, remote procedures calls, data movement, machine failure, creation of new tasks, evaluation of data dependencies and iteration. In contrast to CLOUDSCAPE, these frameworks provide a restricted form of coordinating tasks in a distributed application through the scatter-gather idiom; i.e., they provide a mechanism to map tasks over a number of machines and subsequently gather a result from the machines. Also, the languages of these frameworks describe the control of a system on a central unit, following the workflow approach and so, missing the benefits of the distributed approach as in CLOUDSCAPE.

Typestate for Objects [14] provide a class-based model to declare state transitions as pre- and post-conditions on methods to check invariants on object representation. The language uses a simple syntax of pre- and post-conditions specifications for type-checking clients of classes. Typestate-oriented programming by *Aldrich et al.*[10] provide an object-based model to express state machines. This is an additional mechanism to the object paradigm where objects are modeled in terms of classes and of changing states. Each state may contain a collection of methods associated to it to model an invariant of a class. While, objects in our system contain method calls inside state transition to model control of components. In contrast to CLOUDSCAPE, the models do not allow to express dependencies between two state machines, since this feature is not necessary for the scope of those work.

I/O automata [19] provides a message-passing formalism to model systems of concurrently-operating components. The language of I/O automata is based on "preconditions" and "effects" specifications. In contrast to Typestate and I/O automata, CLOUDSCAPE provides a shared memory model to coordinate and control components of a distributed application, including components added dynamically, based on a language of objects, transitions and dependencies to better structure and re-use the code of management systems.

5. Conclusions and Future Work

This paper presented CLOUDSCAPE, an agent-based language to specify management systems that (1) coordinate and control components of a distributed computation, (2) coordinate and control components added at run-time and (3) restore the normal lifecycle of components due to failure, to provide reliability and scalability of service in the Cloud. The language is based on three simple, elegant idioms, namely object, transition and dependency, to better structure and re-use the code of management systems. It is further extended with non-deterministic update of data attributes to specify component normal lifecycle and its restoration in case of failure. A simple, minimal syntax models the three main idioms of the language. The operational semantics rigorously designs the behaviour of objects and management systems as, respectively, autonomous transitions labeled object and object transitions composed in parallel. We have proven that the operational semantics holds a linear consistent shared memory property. A series of examples illustrate the practical utility and effectiveness of CLOUDSCAPE. Our system follows a distributed approach, where agents themselves structure and share the control on components. While, current management systems follow the centralised approach (workflow), where a central, monolithic unit controls all components of an application. Our approach suits naturally the sort of applications to manage, where each component properties are studied piece by piece, understanding the lifecycle and dependencies, and then specifying the state machines and causalities in CLOUDSCAPE.

We plan to study a static type system that captures meaningless programs formed by a misuse of the language constructs. An interesting aspect to further develop is the parallel composition rule, extending it with two scenarios: (1) two objects can evaluate in parallel if they only read attributes and (2) two objects can evaluate in parallel when they reference the same memory locations, if the attributes read/written or written/written are different. More dynamic concepts such as removal of components are of interest in web services. Another step in developing this work is the implementation of the model as a library of SmartFrog. We plan to identify the communication mechanism between component exceptions and Cloudscape objects. SmartFrog supports an API to access Cloudscape attributes from components code and vice versa; the communication mechanism is based over RMI. The runtime of the library must also support embedding of other mainstream languages such C and C++ to control and coordinate components written in the said languages. We believe that a library that supports the syntax and semantics of CLOUDSCAPE will increase productivity in implementing component-based distributed-applications.

Acknowledgments

We thank Brian Monahan and the anonymous reviewers of CCGrid, FOCLASA and AGERE for comments on a previous version of this paper.

References

[1] Capistrano. Available at `http://www.capify.org/index.php/Capistrano`.

[2] Chef. Available at `http://www.opscode.com`.

[3] ControlTier. Available at `http://controltier.org/wiki/Main_Page`.

[4] Apache Hadoop. Available at `http://hadoop.apache.org`.

[5] HP Server Automation. Available at `https://www.hp.com`.

[6] The Java™ Tutorials: Writing the server side of a socket. Available at `http://java.sun.com/docs/books/tutorial/networking/sockets/clientServer.html`.

[7] Puppet. Available at `http://www.puppetlabs.com`.

[8] SmartFrog. Available at `http://www.smartfrog.org`.

[9] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[10] J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-oriented programming. In S. Arora and G. T. Leavens, editors, *OOPSLA Companion*, pages 1015–1022. ACM, 2009.

[11] A. Bejleri, A. Farrell, and P. Goldsack. *Cloudscape: Language Support to Coordinate and Control Distributed Applications in the Cloud*, 2011. Available at `http://www.doc.ic.ac.uk/~ab406/papers/cloudscape.pdf`.

[12] D. D. Corkill. Collaborating software: Blackboard and Multi-Agent Systems & the Future. In *International Lisp Conference*, 2003.

[13] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[14] R. DeLine and M. Fähndrich. Typestates for objects. In M. Odersky, editor, *ECOOP*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer, 2004.

[15] A. Farrell, P. Goldsack, P. Murray, E. Deloit, and A. Coles. A Hybrid Approach to Autonomic Configuration Management. In *TechCon*, 2009.

[16] K. Fisher, F. Honsell, and J. C. Mitchell. A lambda calculus of objects and method specialization. *Nordic J. of Computing*, 1(1):3–37, 1994.

[17] P. Goldsack, P. Murray, M. Newman, and B. Cox. The Design of a Next Generation Orchestration Engine. In *TechCon*, 2007.

[18] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In P. Ferreira, T. R. Gross, and L. Veiga, editors, *EuroSys*, pages 59–72. ACM, 2007.

[19] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

[20] D. G. Murray and S. Hand. Scripting the cloud with Skywriting. In *HotCloud '10: Proceedings of the Second Workshop on Hot Topics in Cloud Computing*, Berkeley, CA, USA, 2010. USENIX.

[21] A. Omicini, A. Ricci, and M. Viroli. *gens Faber*: Toward a theory of artefacts for mas. *Electr. Notes Theor. Comput. Sci.*, 150(3):21–36, 2006.

[22] A. Ricci, M. Viroli, and A. Omicini. Programming mas with artifacts. In R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, editors, *PROMAS*, volume 3862 of *Lecture Notes in Computer Science*, pages 206–221. Springer, 2005.

[23] A. Ricci, M. Viroli, and G. Piancastelli. simpa: A simple agent-oriented java extension for developing concurrent applications. In M. Dastani, A. E. Fallah-Seghrouchni, J. Leite, and P. Torroni, editors, *LADS*, volume 5118 of *Lecture Notes in Computer Science*, pages 261–278. Springer, 2007.

[24] V. Subramarian. *Programming Groovy: Dynamic Productivity for the Java Developer*. Pragmatic Bookshelf, 2008.