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

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

Andrew Farrell  
*HP Labs, Bristol  
andrew.farrell@hp.com

Patrick Goldsack  
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:

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.

Permission to make digital or hard copies of part of this work for personal or classroom use is granted by ACM, provided that copies are not made or distributed for profit or financial gain, and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © 2011 ACM [to be supplied]. . . $10.00

“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
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.

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 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\footnote{We use the same code as in the Java tutorial [6].}, 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 (sState 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
class Client{
    // including closures, maps, and regular expressions.
    // features of Java and complements it with features from dynamic languages,
    // 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.

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

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

class Client{

    Socket cSocket = null;
    // stream declarations

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

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

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

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 object controls. interact starts the conversation only when the connection with the server is established and updates the status 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 (LOUDSCAPE 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 LOUDSCAPE; 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 LOUDSCAPE 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 2: Client-Server example: Modeling of the agents in CLOUDSCAPE (top) that control and coordinate the behaviour of the Java components (bottom).
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 normally 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 nondeterministic attribute-update. That is, exceptions must be handled 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 state3: 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 exceptions 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.
The agent creates instances of WS, storing the number of Web Server instances (see Figure 5). The constraint describing when to increase the value of the variable stores the number of Web Server instances after the new Web Server is created. Increasing the value of the variable storing the number of Web Server instances only after the create transition in WS takes place 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 response time 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 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.
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. This makes 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 webServerLB mechanism to add dependencies dynamically as explained in Section 2.3.

Propositional expressions include boolean values, data attributes over boolean values & k and L.k (L.k is part of the runtime syntax—syntax of the language introduced at runtime), inequality 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:

- \text{dependency toStart}{\{ lrg.created@src \rightarrow trg \}}
- \text{dependency toCreate}{\{ src.created@src \rightarrow trg \}}
- \text{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 e 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) \rightarrow \{ 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.
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 object 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) \rightarrow \{ created, removed : false;
create : \{\#create, \#create\}\} #create entity
  created : true;
  removed : false\};
remove : \{\#create & \#removed\}\}
  remove : \#create & \#removed\};
  removed : true;
  created : false\}
}
```

\[ D ::= \text{dependency } N[\&/ \{P \leftarrow x \rightarrow y\}] \]
\[ P ::= \text{true} | \text{false} | x.k | Lk | Pop v | P \& P' | P \mid P' \]
\[ e ::= k \mid v \mid e ; e' \mid \text{clone} (e) \rightarrow \{e'\} | N(e, e') \mid e + e' | e e' \mid e' e \mid 0 \]
\[ v ::= \text{object/} \{P\}[e] \mid L \mid n \mid \text{true} \mid \text{false} \]
\[ H ::= L \rightarrow Odesc, H \mid \text{true} \mid \text{false} \]
\[ Odesc ::= Odesc \rightarrow k : v \mid 0 \]
\[ Env ::= N \rightarrow \&/ \{P \leftarrow x \rightarrow y\}, Env \mid 0 \]

Figure 8. User and run-time syntax.

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.

\[ let \, k = e \, in \, e' \Rightarrow k : e' \]

### 3.3 Operational Semantics

Figure 9 gives the operational semantics in a small step style via the reduction relation  \[ \rightarrow \] where the state of the machine is defined only by terms of the calculus and store, written “H; e \rightarrow 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 \[ \oplus \] 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:

\[ H; \text{Main}: P \rightarrow [R-\text{Clone}]^3(k^6) \]
\[ H'; \text{Main}: \text{toStart(unit, L')} ; \text{toCreate(L', L''}); \text{toRemove(L'', L') \rightarrow [R-\text{New}]^2(L)} \]

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:

\[ \text{Object} \mapsto [] \]
\[ L \mapsto [\text{created} : \text{false}, \text{removed} : \text{false}, \text{create} : \text{\{\#create\} \text{created} : \text{true}; \text{removed} : \text{false}], \text{remove} : \text{\{\#create & \#removed\} \text{removed} : \text{true}; \text{created} : \text{false}] \]

\[ L', L'' \mapsto H(L) \cup \{\text{name : A}\}, H(L) \cup \{\text{name : B}\} \]

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

\[ ^6 \text{The number in parenthesis associated to a rule name denotes the number of time that rule is applied.} \]
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:

$$H(L') = Odescr ~ L'' \notin dom(H) ~ H' = H[L'' \rightarrow Odescr \oplus k : v]$$

$$H ; L : \{ k : \text{clone}(L') \rightarrow \{ k : v \}; e' \rightarrow H' ; L : e' \{ L'' / k' \} \} \in dom(H) ~ R - \text{Clone}$$

$$P = \text{Env}(N) \{ L '/ x \} \{ L'' / y \} \transitions(H( L '')) = \{ P' \{ e' \} \}$$

$$H , L : N( L ', L ''); e \rightarrow H , L : (e) L '' : \text{object} \{ P \{ \{ TP'' \{ e'' \} \} \} \} ~ L'' \in dom(H), L'' \notin dom(e) ~ R - \text{New}$$

$$\lor / P' = \text{Env}(N) \{ L '/ x \} \{ L'' / y \}$$

$$H , L : N( L ', L ''); e ; L '' : \text{object} \{ P \{ e'' \} \} \rightarrow H , L : (e)L'' : \text{object} \{ P \lor / \} \{ e'' \} \} ~ R - \text{DepO}$$

$$H , L : \text{Main} : \{ L_1 : \text{object} \{ P_1 \{ e_1 \} \} \mid \ldots \mid L_n : \text{object} \{ P_n \{ e_n \} \} \rightarrow H , L : \text{object} \{ P_1 \{ e_1 \} \mid \ldots \mid L_n : \text{object} \{ P_n \{ e_n \} \} \} ~ R - \text{SpawnerM}$$

$$\forall j \in \{ 1 \ldots n \}, \text{eval}(H(L), P_j) = \text{false} \Rightarrow \text{eval}(H(L), P_j) = \text{true}$$

$$\forall j \in \{ 1 \ldots n \}, \text{eval}(H(L), P_j) = \text{false}$$

$$H , L : \{ P_j \{ e_j \} \}_{j \neq i} : \text{object} \{ P \{ e \} \} \rightarrow H \{ \} \in dom(H) ~ R - \text{TranT}$$

$$\forall j \in \{ 1 \ldots n \}, \text{eval}(H(L), P_j) = \text{false}$$

$$H , L : \text{object} \{ P \{ e \} \} \rightarrow H \{ \} \in dom(H) ~ R - \text{TranF}$$

$$k \in \text{dom}(H(L)) \Rightarrow H' = H(L)[k \rightarrow v]$$

$$H , L : k : e ; e' \rightarrow H ; L : e'$$

$$H_1 : e \rightarrow H_1' ; e'$$

$$H_2 ; g \rightarrow H_2' ; g'$$

$$\text{dom}(H_1') \cap \text{dom}(H_2') = \emptyset$$

$$\text{dom}(H_1) \cap \text{dom}(H_2) = \emptyset ~ R - \text{Par}$$

$$H_1 , H_2 ; g \rightarrow H_1' , H_2 ; g'$$

$$H_1 ; e \rightarrow H' ; e'$$

$$H ; e \rightarrow H' ; e'$$

$$H , l : e \rightarrow H , l'$$

$$H , l : e \rightarrow H , l'$$

Figure 9. Operational Semantics

Spawning behaviors. Rule R-SpawnerM 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:

$$H' ; L : \text{object} \{ L' \text{.create} \} \{ L'' \text{.remove} \} \{ create : \ldots ; remove : \ldots \}$$

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

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
The value of an object’s attribute can be updated from the scope of a local transition associated to the object through rule \( \text{R-Attribute} \). The expression \( \{ e \mid v \} \) evaluates the expression \( e \) to the value \( v \).

In the example, \( L' \) updates the attribute \( created \), resulting in:

\[
H''; L': \text{create} : \ldots; \text{remove} : \ldots; \text{object} [ L'.\text{created} ] \quad \text{with} \quad H''; \text{remove} [ L'.\text{created} ]; \text{create} : \ldots; \text{remove} : \ldots; \quad [\text{R-IPar, R-ObjectT}]
\]

The computation follows on the data-attribute update chosen. **Parallel evaluation.** The rule \( \text{R-Par} \) 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 \( \alpha \) 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 \( \alpha \)-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 \( \alpha \) 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 parallel, partitioning the memory to satisfy the side conditions 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 \( \text{R-TranT} \), resulting in:

\[
H; L': \text{create} : \text{true}; \text{remove} : \text{false}; \text{object} [ L'.\text{created} ]; \text{remove} : \ldots; \text{object} [ L'.\text{created} ]; \text{create} : \ldots; \quad [\text{R-IPar, R-ObjectT}]
\]

**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 \( H; e \rightarrow H'; e' \Rightarrow \overline{T} \). The full definition and proofs are given in a companion technical report [11].

**Lemma 3.1.** If \( H; e \rightarrow H'; e' \Rightarrow \overline{T} \) then \( \{e\} \subseteq \text{dom}(H) \).

**Theorem 3.2 (Linear consistent memory).** If \( H; e \rightarrow H'; e' \Rightarrow \overline{T} \) then \( \overline{T} \) has no duplicates.

**Proof.** By induction over the operational semantics rules and Lemma 3.1. \( \square \)

### 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 \( Q \) respectively denote the clone expressions for the client and server object in Figure 2.

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

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

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

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

\[
\begin{align*}
L & \rightarrow \square \\
\text{Object} & \rightarrow \square \text{[address : "localhost", port : 1234, cState : "raw", connect : \{cState = "raw"\}, interact : \{cState = "connectionEst"\}, close : \{cState = "completed"\}]}
\]

\[
L' & \rightarrow \square \text{[port : 1234, sState : "raw", listen : \{sState = "raw"\}, accept : \{sState = "ready"\}, interact : \{cState = "connectionEst"\}, close : \{cState = "completed"\}]}
\]

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

where \( Q' \triangleq \text{object [true] \{listen : \{sState = "raw"\}, accept : \{sState = "ready"\}, interact : \{sState = "connectionEst"\}, close : \{sState = "completed"\}\}} \)

\[
H''; \text{Main}: (L' : Q' \mid L : P') - [R-\text{SpawnM}]
\]

where \( P' \triangleq \text{object [L'.sState \neq "raw"\{connect : \{cState = "raw"\}, interact : \{cState = "connectionEst"\}, close : \{cState = "completed"\}\}} \)

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

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

\[
\emptyset, H''; L' : Q'; Q' \mid L : P' - [R-\text{IPar}, \text{R-TranT}]
\]

where \( Q'' \triangleq \text{listen: \{sState = "raw"\}, accept: \{sState = "ready"\}, interact: \{cState = "connectionEst"\}, close: \{sState = "completed"\}\}

\[
H''; L' : \text{server theServer = new Server(port); sState : "ready"; Q'} \mid L : P' - [R-\text{IPar}, \text{R-Attribute}]
\]

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

where \( H''' \subseteq H''(L') \cup \{\text{sState : "ready"}\} \)

\[
\emptyset, H''' ; L' : Q''; Q' \mid L : P'' - [R-\text{IPar}, \text{R-TranT}]
\]

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 curious 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 &kA(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. &ampa [23] is a Java library that uses the &kA meta-model to provide support for designing the architecture of multi-threaded/concurrent applications. Agents in &ampa 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 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


