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

Andi Bejleri, Andrew Farrell, Patrick Goldsack

HP Laboratories
HPL-2010-197

Keyword(s):

Coordination, Control, Cloudscape, Smart Frog

Abstract:

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. 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, multi-tenancy, and service heterogeneity. This paper proposes an object-based language extended with dependencies, 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. The problem context is further extended with component failure and dynamic addition of new components. Our language allows programmers to write the dependencies between the lifecycle states of components as relations between the language objects that are responsible for controlling components behaviour.

External Posting Date: December 6, 2010 [Fulltext] Approved for External Publication
Internal Posting Date: December 6, 2010 [Fulltext]

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

Andi Bejleri*
Department of Computing
Imperial College
London, UK
Email:ab406@doc.ic.ac.uk

Andrew Farrell
and Patrick Goldsack
*HP Research Labs
Bristol, UK
Email: <name>.<lastname>@hp.com

***Abstract*—*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. 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, multi-tenancy, and service heterogeneity.**

This paper proposes an object-based language extended with dependencies, 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. The problem context is further extended with component failure and dynamic addition of new components. Our language allows programmers to write the dependencies between the lifecycle states of components as relations between the language objects that are responsible for controlling components behaviour.

I. 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, multi-tenancy, 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, for example, a service in a Web Services or a task in parallel algorithms and scientific computations.

Components of a distributed application define *dependencies at different states of their lifecycles*. For example, in a basic Client-Server program [8], the running order of the two components is defined, citing the Java Tutorial [8], 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. In addition, failure coming from components logic or hardware, which affects the normal lifecycle of components, becomes common place. Thus, we would like a language that answers this research question:

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

The solution proposed in this paper is an object-based language extended with dependencies and non-deterministic update, called CLOUDSCAPE. An *object* describes the abstract state machine of a component lifecycle and a *dependency* describes a causality between the states of two objects. Non-deterministic update is used to describe scenarios where state can change normally to the next one, according the logic of the component, or exceptionally due to failure. An object is not only a blue-print of a component lifecycle but also a machine interpretation of it. That is, an object controls the behaviour of a component through the associated *transitions* that perform the change of object's state. While methods in OO languages are a block of statements, CLOUDSCAPE transitions are block of statements guarded by a predicate. Constraints are enforced at runtime, where a transition defined on a state that is dependent on a state of another object takes place only when that dependency is true.

Another problem is to coordinate and control new components added at run-time 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 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 dynamically added components?*

The solution to this problem is providing CLOUDSCAPE with the feature of adding new objects from the body of transitions.

Our solution to both the questions follows a distributed approach, where objects 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 [4] and Capistrano [2], where a central, monolithic unit controls all the components of an application. Our distributed approach suits naturally the sort of applications we are trying to manage, where each application component properties are studied piece by piece, understanding their lifecycles and dependencies, and then building the state machines and causalities between states in CLOUDSCAPE. We have experienced that managing large-scale systems following the distributed approach leads to more readable, robust and scalable systems than following the workflow approach.

Organization. The remainder of this paper is organized as follows. Section II gives the intuition of the language throw two real-world examples: Client-Server and Load Balancer. Section III discusses our syntax and operational semantics, illustrated by a 2-component example. Section IV illustrates how our system prevents data races when evaluating global predicates through the Dinning Philosophers example. Section V surveys related work and section VI concludes with a discussion of possible future work for this system.

II. 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. Before giving the examples, we give the definition of the three main constructs in CLOUDSCAPE, namely object, transition and dependency.

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

Definition 2 (CLOUDSCAPE transition)**.** *A transition consists of a block of programming statements and a predicate. The block of statements performs actions in an external language (Java in our study) and* CLOUDSCAPE*. The predicate guards the run of the block of statements. Transitions provide a mechanism to control the behaviour of a component as they change the state of the object they are associated with.*

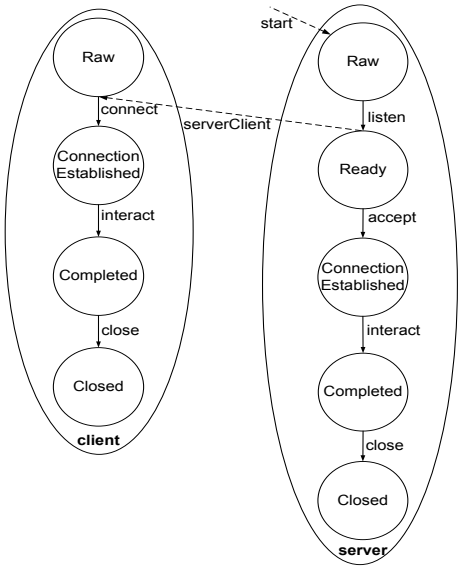

Fig. 1. State diagram of Client-Server

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, where the second parameter, denoted* `by`*, represents the object that the propositional expression will guard, called the* dependent object *while the first object is called the* depending object*. Dependencies provide a mechanism to enforce the constrains between the lifecycle states of two objects and to define object's runnability.*

A. Client-Server Example

Figure 1 gives the state diagram of the Client-Server example, described in the introduction. The system consists of two objects `client` and `server`, and two dependencies `start` and `serverClient`. A server is initially created listening for connection from the client. Once a request for connection has arrived, the server accepts it establishing a connection with the client. Further, the server interacts with the client and subsequently completes its run. Lastly, the server closes all the streams and sockets opened ahead. Whilst this is taking place, a client is created connecting to a listening server. Once the connection is established, the client interacts with the server. Similarly to the server, the client ends by closing all the streams and sockets opened when the connection was established.

The implementation in CLOUDSCAPE of the diagram is shown in Figure 2. Each object controls respectively the behaviour of each Java component: `Client` and `Server`. For presentation reasons, we omit the full component's code[1], relegating it to Appendix. An object is created by cloning

[1]We use the same code as in the Java Tutorial [8].

```
dependency start{true@on→by}
dependency serverClient {
                (on:sState≠"raw")@on → by}

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

                connect [cState="raw"]{
                  Client theClient =
                    new Client(address, port);
                  cState "connEstbl"
                 };
                interact [cState="connEstbl"]{
                  theClient.interact();
                  cState "completed"
                 };
                close [cState="completed"]{
                  theClient.close();
                  cState "closed"
                 };
                },
    server = clone(Object)↔{
                port 1234;
                sState "raw";

                listen [sState="raw"]{
                  Server theServer =
                     new Server(port);
                  sState "ready"
                 };
                accept [sState="ready"]{
                  theServer.accept();
                  sState "connEstbl"
                 };
                interact [cState="connEstbl"]{
                  theServer.interact();
                  cState "completed"
                 };
                close [cState="completed"]{
                  theServer.close();
                  cState "closed"
                 };
                }
in
    start(unit, server);
    serverClient(server, client)
```

Fig. 2: Client-Server example: the CLOUDSCAPE objects control the behaviour of the Java components (see Figure 3)

```
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 declarations

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

Fig. 3: Client-Server example: Java components

another object, `Object` by default, updating attributes of the cloned object and adding new attributes. `server` and `client` contain attributes to set up the components, e.g. `address` and `port`, and attributes that store the state of component's lifecycle, e.g. `cState` and `sState`, initially set to *Raw*. Transitions are defined strictly on the component's own state. The `connect` and `listen` transitions in `client` and `server`, respectively, occur only if the state of the components is *Raw*. Furthermore, `connect` creates an instance of the `Client` class, and updates the state attribute with the new component's state *Connection Established*. Java code can be embeded in CLOUDSCAPE by injecting Groovy scripts— Groovy [17] 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. The constructor code of `Client` creates a connection with `Server`. In the `Server`, `listen` creates an instance of the `Server` class, and updates the state attribute to *Ready*. The constructor of `Server` creates a server socket ready to accept connections from `Client`. In `client`, `interact` controls the interactions with the server by invoking the `interact` method on the `Client` instance. `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 closing all the streams and sockets opened ahead. In the server side, `accept` controls a ready `server` to accept a connection, where a ready `server` is defined over a listening server socket. `interact` and `close` control the computation of the `Server` similarly as the ones of the `Client`.

The behaviour of `client` is guarded by the `serverClient` dependency. That is, `client` is active only if `server` is not in the *Raw* state. While, the behaviour of `server` is guarded by a dependency that does not enforce any constraint but 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 lacking

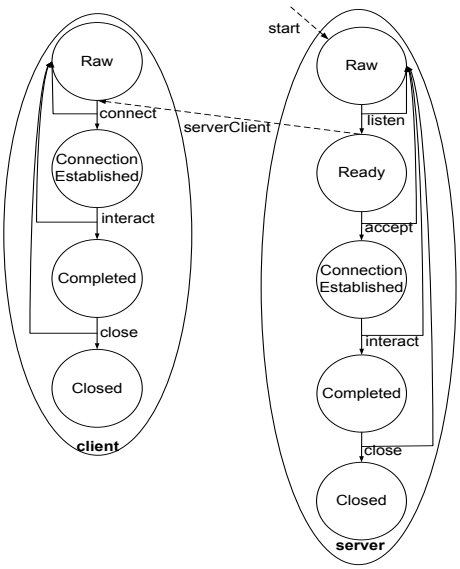

Fig. 4. State diagram of Client-Server handling failure

of depending object, in a similar sense as the "void" type in Java denotes lack of returning output. We use the term "unit" from ML. Hence, the behaviour that invokes starting the client will be applied after the server is listening for connections. The constraint described in the tutorial is expressed as a propositional expression on the state of components, guarding the behaviour of an object.

B. Client-Server Example with Failure

Every transition of each object may fail, transiting the state of an object to *Raw*[2] as shown in Figure 4. In case of an erroneous action caused by component logic or hardware failure, the Java runtime engine interrupts the program, affecting the normal lifecycle of the component and possibly of all the other components of an application.

The Java runtime provides a mechanism to handle erroneous actions through the `try − catch` clause. In the example of the previous section, exceptions were handled at the component level, interrupting the running of one component, and consequently of the other, without notifying the CLOUDSCAPE objects. As a consequence, the state machine represented in the CLOUDSCAPE object is erroneously active on an abstract component state that does not match the real one.

Figure 5 shows how to restore the computation in the Client component (Client-Server example) in case of failure using CLOUDSCAPE. Our solution shifts the handle of exceptions at the CLOUDSCAPE objects and so, updating the attributes that abstract the state of components to *Raw*. That is, the state machine of a component is set to *Raw* if an exception is raised when executing part of the component behaviour. In the Client-Server example, if an exception is raised in one of the methods `connect`, `listen`, `accept`, `interact`, and `close` that are

[2]The design to handle failure by transiting the state to the initial one is related to this particular example and should not be considered as a design pattern on how design failure handling in CLOUDSCAPE.

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

            connect [cState="raw"]{
                try{
                    Client theClient = new
                      Client(address, port);
                    cState "connEstbl"
                }catch(UnknownHostException e){
                    ... /* handle exception for
                           component */
                    cState "raw"
                }catch(IOException){
                    ...
                    cState "raw"
                }
            };
            interact [cState="connEstbl"]{
                try{
                    theClient.interact();
                    cState "completed"
                }catch(IOException e){
                    ...
                    cState "raw"
                }
            };
            close [cState="completed"]{
                try{
                    theClient.close();
                    cState "closed"
                }catch(IOException e){
                    ...
                    cState "raw"
                }
            };
        }


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 */
}
```

Fig. 5: Client-Server example: Client—Handling component failure in the object and not in the component

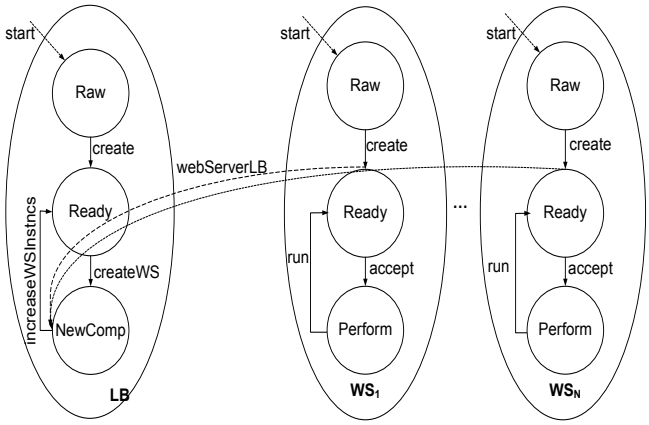

Fig. 6. State diagram of Load Balancer

controlled by the transitions of the same names, the latters restore the computation of each component at the initial state. A restore of the computation of one participant will cause the other participant to restore the computation to the initial state, leading the two components to the initial state *Raw*.

C. Load Balancer

The Load Balancer manages and dispatches the load of work to several Web Servers in relation to time response. That is, it adds a new Web Server into the session if the response time is lower than a threshold and then dispatches the work load to each of them. For presentation reasons, we have simplified the specification and state diagram of the problem to only the dynamic features of it. The state diagram, given in Figure 6, consists of the `LB` object and the `WS` objects, and dependencies `start` and `webServerLB`. The transitions of the `LB` object flow over three states: *Raw*, *Ready* and *NewComp*, where the cycle between states *Ready* and *NewComp* creates new web server objects and addresses the load of work to the new web servers. The transitions of the `WS` objects are defined over the *Raw*, *Ready* and *Perform* states, where `create` creates an instance of a Web Server, `accept` delegates to the instance created a load of work to perform and `run` accomplish the processing of the work. A web server is always accepting requests of work and processing them, represented in the diagram through a cycle between the *Ready* and *Perform* states. The work is dispatched to the new Web Server only when the latter have been created and are ready to accept requests of work. Figure 7 gives the modeling of this specification in CLOUDSCAPE.

The CLOUDSCAPE object that describes the state machine of the Load Balancer, firstly, starts the Load Balancer component and then, repeatedly adds a new CLOUDSCAPE object responsible for a Web Server, if the response time of a service is lower than a certain threshold. The work to the new Web Server component is dispatched if the Web Server has been created by the object. The state machine of a Web Server, described in the `WS` object, firstly, creates a Web Server component and then starts it. The constraint that part of the work load is dispatched to the new Web Server only if it has been created, is defined as a dependency on the state machine

```
dependency start{true@on → by}
dependency webServerLB {
            &(on:wsState="created")@on → by}

let LB = clone(Object)↔{
        respTime 0;
        threshold 1000;
        wsInstncs 0;
        lbState "raw";

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

    WS = clone(Object)↔{
        wsState "raw";

        create [wsState="raw"]{
                WebServer ws = new WebServer();
                wsState "ready"
        };
        accept [wsState="ready"]{
                ws.accept();
                wsState "perform"
        };
        run [wsState="perform"]{
                ws.run();
                wsState "ready"
        }
    }
in
   start(unit, LB)
```

Fig. 7: Load Balancer example: CLOUDSCAPE objects

of the Load Balancer `webServerLB`, i.e. the state machine becomes inactive to transit from the state `newComp` to `ready` through `run` if the dependency added at `addWS` is not true. The `start` dependency is used in the same way as in the Client-Server example, to start each component.

The attribute `respTime` stores the response time of a service and is updated in the Load Balancer component code. The `wsInstncs` attribute stores the number of web server created and its value is used by Load Balancer component code when dispatching the workload. Section V explains how CLOUDSCAPE attributes values can be read and written from Java code in our study.

III. Formal Model

We now introduce the core CLOUDSCAPE language to formalize the intuitions given above. This section contains the full syntax and operational semantics.

A. Syntax

Figure 8 provides the syntax of our language. The metavariable N ranges over dependency names; `on`, `by` range over dependency variables; e, e', g, ... range over expressions; x, y, ... range over variables; k, l, ... range over attributes names; L, L', L'', ... ranges over locations.

A program is a list of objects, followed by a list of instances of dependencies. Dependencies are defined over two objects bound by the @ operator. Dependencies in CLOUDSCAPE are defined over any propositional expression to capture states of objects as shown in the above examples. In CLOUDSCAPE, dependencies are composed by the & and | operators to compose different states of the same object through | and different states of different depending objects through &. Proposition expressions include boolean values, in- equality tests ($<$, $>$, =) on attribute values of objects, and two expressions composed using the boolean operators `and`, denoted &, `or`, denoted |, and `not`, denoted !.

Expressions define behaviour in CLOUDSCAPE. An attribute is defined over a name and value. An object is created by cloning another object, Object by default, and update it with new attributes similarly as in the system of Fisher *et al.* [12]. An instance of a dependency is created when two objects are applied. The sequential composition is standard. A data attribute can be updated non-deterministically by two values to represent a normal transition of one state to another following the normal flow of the component and an exception transition due to component failure. Values include transitions and primitive values such as natural numbers and boolean values.

Transitions labeled object represent the guarded behavior of an object, where dependencies' instances upon the object define the predicate (the guard) and the transitions(behaviour) associated to the object define the scope of the expression (the 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 components attributes (global), while in transitions, predicate is defined over component's own data attributes (These constraints are ensured by the operational semantics in Figure 9).

The remaining constructs are part of the runtime syntax. Location of an object and parallel composition is standard, composing in parallel the behaviour of two or more objects. At run-time, the object behaviour are prefixed by the object location to define rules of scoping when new objects are created as we shall see later in the operational semantics.

Objects's attributes are stored in a heap that is a pair of object location and description. The object description is a sequence of attributes. Our attribute-based object encoding is similar to standard object encodings [9], [12].

$$
\begin{array}{lcl}
D & ::= & \textsf{dependency N}\{\&/|/\ ^{3}P@\text{x}\rightarrow \text{y}\} \\
P & ::= & \textsf{true} \mid \textsf{false} \mid \text{x:k} \mid \text{L:k} \mid P\ \textsf{op}\ \text{v} \\
 & & \mid P\&\ P' \mid P \mid P' \mid\ !P \\
e & ::= & \text{k}\ e \mid \textsf{clone}\ (e) \leftrightarrow \{e'\} \mid \text{N}(e, e') \\
 & & \mid e;e' \mid e \oplus e' \mid v \mid L \mid e \mid e' \mid \text{L:}e \\
v & ::= & \textsf{object/}\ ^{4}\ [\text{P}]\{e\} \mid \text{n} \mid \textsf{true} \mid \textsf{false} \cdots \\
H & ::= & L \rightarrow Odescr,\ H \\
Odescr & ::= & Odescr \leftrightarrow \text{k}\ e
\end{array}
$$

Fig. 8. User and run-time sytax

a) Encoding of the let *construct:* In Section II, 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 sequential composition constructs as shown below.

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

B. Operational Semantics

Figure 9 gives the operational semantics via the reduction relation $\longrightarrow$ where the state of computation is defined by terms of the language and a heap of objects. The interesting features of the rules are how they initiate a session, create a new object in the heap and in the evaluation scope, add a dependency to an object, evaluate an object transition and a transition associated to an object behavior, update non-deterministically an attribute, and prevent race conditions regarding evaluation of global predicates.

Rule *R-Init* initiates a session by introducing the Main reference to the main program that is defined as a list of object declarations and instances of dependencies on those objects. Once the session has been initiated, new objects are created in the heap through rule *R-Cloning*. In the heap, the new object contains the attributes of the cloned object and the ones added by the user. The attributes of the cloned object that have the same name to the user ones are replaced by the latter using the $\uplus$ operator defined in Figure 10.

The behaviour of a new object (object transition) is created only when the first dependency is applied to him. Rule *R-New* places the new behavior inside the scope of the evaluating object, prohibiting it to run until the running transition has terminated. This design allows to capture all the dependencies instances on the created object that could be present in the running transition before making the behaviour of the object runnable. `Env` defines the environment of dependencies for a program. The dependency returned is instantiated by replacing `x` with the first argument and `y` with the second argument in the propositional expression. `transitions` looks up the object descriptor for transitions (see Figure 10). Rule *R-DepO* applies an instance of dependency to an object. The instance is

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

[4] object/ denotes either object or none.

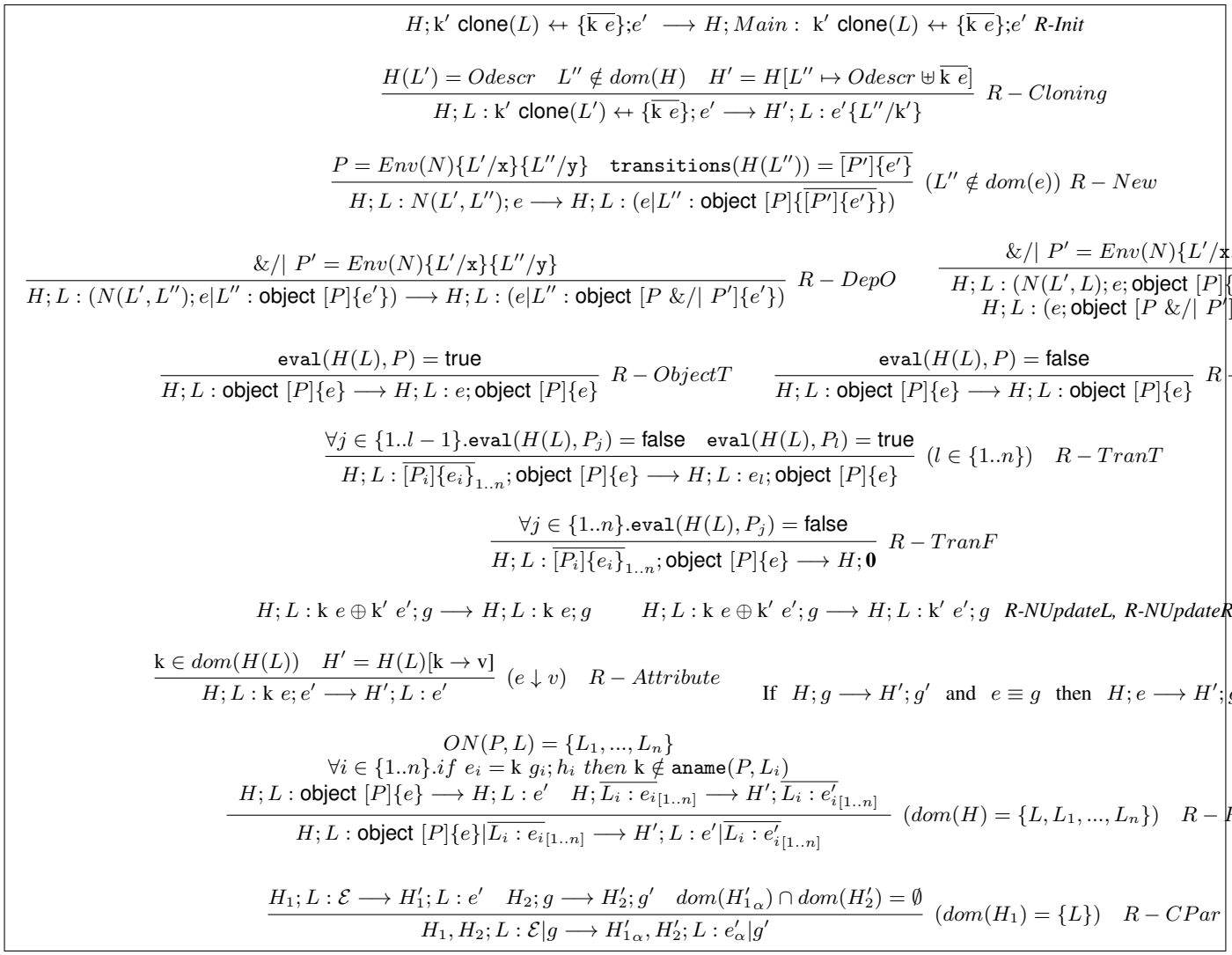

Fig. 9: Operational Semantics

created similarly as in *R-New* and is added to the guard of the second object's behaviour, following the composibility rules of | or &. Other rules add a dependency to the running object (self) by adding it to the guard of object behaviour after the current transitions has been executed (*R-DepS*).

An object transition evaluates the block of expressions (transitions associated to the object), concatenated to the object behavior to provide continuity of computation, if the global predicate evaluates true (rule *R-ObjectT*), otherwise it reduces to itself. This contrasts the semantics of transitions where rule *R-TranT* looks up in the list of transitions associated to the object for a transition that predicate evaluates to true, returning the block of statements of the latter. Rule *R-TranF* signifies the end of an object's behavior since no transition is available to run; i.e. all transitions predicates evaluate to false. Both these rules allow predicates to be defined strictly over object's attributes by restricting the scope of the heap H to $H(L)$ when evaluating the predicates (see Figure 10). A clear separation at the language definition between the kind of attributes used in dependencies and guards of local transitions educates programmers to use the state machine metaphor while modeling lifecycle management of systems, where local transition defines a change of lifecycle state of a component and dependency defines a causality between two lifecycle states of two components.

The value of an attribute can be updated from the scope of a local transition. The rule *R-Attribute* allows change to attributes that are part only to the current objects's attributes. The expression $(e \downarrow v)$ denotes the evaluation of the expression e to the value v, where v denotes primitive values and transitions. Rules *R-ChoiceL* and *R-ChoiceR* represent the non-deterministic choice on the two data attribute updates, respectively the left and right attribute update. The computation follows on the data attribute chosen. Rule *R-Congr* spawns the behavior of new objects according to structural congruent rules given below:

L: (object [P]{e}|L_1:object $[P_1]\{e_1\}$ | ... |L_n:object $[P_n]\{e_n\}$)≡
L:object [P]{e}|L_1:object $[P_1]\{e_1\}$ | ... |L_n:object $[P_n]\{e_n\}$

Main:(L_1:object $[P_1]\{e_1\}$ | ... |L_n:object $[P_n]\{e_n\}$)≡
L_1:object $[P_1]\{e_1\}$ | ... |L_n:object $[P_n]\{e_n\}$

Union of attributes

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

Transitions look up

$\texttt{transitions}(Odescr \leftarrow \mathrm{k}\ e) = \texttt{transitions}(Odescr) \cup \texttt{transitions}(e)$
$\texttt{transitions}([P]\{e\}) = \{[\mathrm{P}]\{\mathrm{e}\}\}$ $\quad \texttt{transitions}(e) = \emptyset, e\ notInstanceOf\ Transition$

Evaluation of predicates

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

Depending object look up

$ON(\mathsf{true}, L) = \emptyset$ $\quad ON(\mathsf{false}, L) = \emptyset$ $\quad ON(L : x, L) = L$ $\quad ON(L' : x, L) = \emptyset$
$ON(P\ \mathsf{op}\ v, L) = ON(P, L)$ $\quad ON(P\&/|P') = ON(P, L) \cup ON(P', L)$
$ON(!P, L) = ON(P, L)$

Attribute names look up

$\texttt{aname}(L : \mathrm{k}, L) = \{\mathrm{k}\}$ $\quad \texttt{aname}(L' : \mathrm{k}, L) = \emptyset$
$\texttt{aname}(P\&/|P', L) = \texttt{aname}(P) \cup \texttt{aname}(P')$ $\quad \texttt{aname}(!P, L) = \texttt{aname}(P)$
$\texttt{aname}(\mathsf{true}/\mathsf{false}/x : \mathrm{k}, L) = \emptyset$

Fig. 10: Auxiliary definitions

where the first rule spawns the behaviors after the transition that has generated them in L_1 is reduced to the object transition, following rule *R-ObjectT*, and the second rule spawns the behaviors after the main program has been evaluated.

The dependencies of an object (global predicate) are evaluated only if the depending objects are not updating in parallel the attributes that define those dependencies. This is to prevent race conditions between the dependent and several depending objects where the former is reading and the latters are writing on the same attributes. In the rule *R-Par*, function $ON(P, L)$ looks up for depending objects of L by scanning the global predicate P for references different from L (see Figure 10). The second condition checks the scope of the runnable objects whether the depending objects are updating attributes; in that case, the condition is satisfied only if the attributes are different from the ones in dependencies, resulting in the evaluation of the object transition in parallel with the evaluation of the other running objects. `aname` looks up for attribute names in the part of proposition P defined by L's attributes (see Figure 10). Rule *R-CPar* defines how execution of other expression, different from object transitions, occur in parallel composition. The formal definition of $\mathcal{E}$ is given in Appendix. The reduction of these expressions occur independently with a memory of only the object they are associated to. The new heap returned by the first sub-reduction is refreshed with new reference names in the scope, so that the new reference names created in the heaps H_1' and H_2' do not clash when they are combined together. Refreshing of reference names is defined through α-conversion as in the lambda calculus. This allows to define a parallel composition rule that preserves a consistent shared memory.

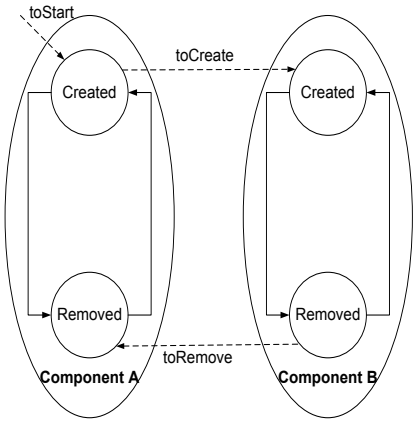

Fig. 11. State diagram of Two Components

C. Example of Two Components

We illustrate how the formal model of this work can coordinate and control a system of two components, namely `A` and `B`. The logic of each component consists of creating and removing an entity (e.g. a virtual machine). The specification of the system defines component `A` to create first an entity followed by the creation of a second entity by `B`. Component `B` can create and remove an entity repetitively in the system until component `A` has removed its entity. The last action takes place only when `B` has removed its entity.

Figure 11 shows the state machine of the two component system. Each component includes two states, namely `Created` and `Removed`. The dashed arrows define the dependencies between the two states of components. The `toStart` dependency starts the session, while dependencies `toCreate` and `toRemove` define respectively the order of creating and removing entities between components `B` and `A`.

Below, we provide the CLOUDSCAPE implementation of the said state machine:

```
dependency toStart{ !by : created@on→ by}
dependency toCreate{on : created@on→by}
dependency toRemove{ | (on : removed)@on→by}

let O = clone(Object) ↔ {
                    created false;
                    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 objects A and B control respectively components `A` and `B` and, dependencies $toStart$, $toCreate$ and $toRemove$ capture the dependencies defined in the specification. The $toStart$ dependency is defined on the initial state of A, where the attribute `create` is set to false, and is used to start the computation of A. The $toCreate$ dependency captures the state of the dependent (A) as "created" and applies to B. The $toRemove$ dependency captures the state of the dependent (B) as "removed" and applies to A. In the *create* and *remove* transitions, the line starting with # defines the invocation of respectively creating and removing an entity. For presentation reasons, we omit that line from the definition of objects in the remaining of this section. Some of the reduction steps of the program are given below:

$\emptyset$; P $\longrightarrow^{[R-Init]}$

$\emptyset$; Main: P $\longrightarrow^{[R-Cloning](3)}$[5]

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

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

H; Main: $(L'$:object$[!L'$:$created \mid L''$:$removed]\{create$ ...;
$remove$ ...$\}$
$|L''$:object$[L'$:$created]\{create$ ...; $remove$ ...$\})$
$\longrightarrow^{[R-Congr,R-Par(2),R-ObjectT,R-ObjectF]}$

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

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

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

where P denotes the program (objects and dependency instances). The evaluation of the program P starts with an empty heap. New objects are created in the heap following rule *R-Cloning*, defined as follows:

$$
\begin{array}{lcl}
L & \mapsto & [created \text{ false}, removed \text{ false}, \\
 & & create\ [!created]\{created \text{ true}; removed \text{ false}\}, \\
 & & remove\ [created\&!removed]\{removed \text{ true}; \\
 & & \qquad created \text{ false}\}] \\
L' & \mapsto & [created \text{ false}, removed \text{ false}, name\ A, \\
 & & create\ [!created]\{created \text{ true}; removed \text{ false}\}, \\
 & & remove\ [created\&!removed]\{removed \text{ true}; \\
 & & \qquad created \text{ false}\}] \\
L'' & \mapsto & [created \text{ false}, removed \text{ false}, name\ B, \\
 & & create\ [!created]\{created \text{ true}; removed \text{ false}\}, \\
 & & remove\ [created\&!removed]\{removed \text{ true}; \\
 & & \qquad created \text{ false}\}]
\end{array}
$$

In the third step, the behavior of A (referenced by L') guarded by dependency $toStart$ is placed in the scope of Main, disallowing it to run (by rule *R-New*). After all the dependency instances have been evaluated, the two behaviors are spawned to run independently according to the second congruence rule discussed in Section III-B. Both behaviors of the two objects are evaluated in parallel, each following rule *R-Par*, resulting in the evaluation of L' transitions as L'' global predicate evaluates to false. L' evaluates the first transition through rule *R-CPar* in parallel with the evaluation of L'' global predicate through rule *R-Par*. In the next step when L' updates the attribute *created*, L'' can not evaluate its global predicate in parallel since rule *R-Par* can not take place as the (second) condition that checks for race conditions

[5]The number in parenthesis denotes the number of time that rule is applied.

fails. The new heap H' reflects the update of L',

$$\begin{array}{lll} L' & \mapsto & [created\ \mathsf{true}, removed\ \mathsf{false}, name\ A, \\ & & create\ [!created]\{created\ \mathsf{true}; removed\ \mathsf{false}\}, \\ & & remove\ [created\&!removed]\{removed\ \mathsf{true}; \\ & & \qquad\qquad created\ \mathsf{false}\}] \end{array}$$

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

IV. Dinning Philosophers Problem

We investigate deadlock in our calculus through a classic concurrency problem—Dinning Philosophers [16]. Each of the five philosophers is eating or thinking, without overlapping the two activities. In order to eat each philosopher needs two forks and there are as many forks in the table as the number of philosophers.

The solution of the problem in our calculus is given in Figure 12. A deadlock situation can occur when each of the philosophers picks one of the forks and waits continuously until one of the neighbours releases a fork. In our solution, a philosopher is active when both forks are available; that is the case when the two neighbours are either in the *thinking* or *hungry* states. The *Neighbour* dependency captures the condition of using two forks through the `and` (&) operator. There are no race conditions when the dependencies are evaluated on each object as they are evaluated in order and not simultaneously. Also, no data corruption can occur between two neighbours, in the scenario when one component evaluates its global predicate whilst one of its neighbours is updating it. By our semantics, an object can evaluate its behaviour if none of the dependent objects that define the global predicate are updating the attributes that global predicate is using.

Our solution is distributed; i.e., every philosopher checks the state of his neighbours through evaluation of the dependencies assigned to him. There is no central entity as in the Waiter solution where the Waiter has a global view of the forks available during the dinner and manages the requests from Philosophers to avoid deadlocks. Our solution is similar to the Monitor one in that, an object that cannot get the second fork must put down the first fork before they try again. Each object can access the global state through a lock on a procedure, our evaluation of dependencies. However, our solution reduces concurrency in this system, compared to the monitor solution as a philosopher can not change the state, that is independent from the state of the neighbours, from *thinking* to *hungry* if one its neighbours is changing its state.

$$\begin{array}{l} \mathsf{dependency}\ Neighbour\{\&(on.state = THINKING \\ \qquad |\text{on.state=HUNGRY})\} \\ \mathsf{dependency}\ Own\{|\ \text{on.state=THINKING}\} \\ let\ Philosopher = \mathsf{clone}(Object) \leftarrow \{ \\ \qquad state\ THINKING; \\ \qquad eat\ [state = HUNGRY]\{state\ EATING\} \\ \qquad think\ [state = EATING]\{state\ THINKING\} \\ \qquad hungry\ [stateTHINKING]\{state\ HUNGRY\} \\ \qquad \} \\ in \\ \quad let\ phil1, phil2, phil3, phil4, phil5 = \mathsf{clone}(Philosopher) \\ \quad in\{ \\ \qquad Neighbour(phil2, phil1); \\ \qquad Neighbour(phil5, phil1); \\ \qquad Own(phil1, phil1); \\ \qquad Neighbour(phil1, phil2); \\ \qquad Neighbour(phil3, phil2); \\ \qquad Own(phil2, phil2); \\ \qquad Neighbour(phil2, phil3); \\ \qquad Neighbour(phil4, phil3); \\ \qquad Own(phil3, phil3); \\ \qquad Neighbour(phil3, phil4); \\ \qquad Neighbour(phil5, phil4); \\ \qquad Own(phil4, phil4); \\ \qquad Neighbour(phil4, phil5); \\ \qquad Neighbour(phil1, phil5); \\ \qquad Own(phil5, phil5); \\ \qquad \} \end{array}$$

Fig. 12. Our solution of the Dinning Philosophers problem

V. Related Work

SmartFrog The idea of dependency modeling in CLOUDSCAPE originates from previous works by the authors on management of federated systems [13] 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 [7] 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 [10] — 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. While the blackboard metaphor consists also of an arbiter that decides which object to run in the case when more than one object is active, CLOUDSCAPE semantics is not defined over a centralised running entity that controls the objects but rather every object is a running entity that runs independently and communicates with other objects through dependencies.

Despite its maturity, SF does not support coordination and control of components in a distributed application and so, leaving unsolved the two questions of this paper. However, SF offers a nice platform to implement and further develop CLOUDSCAPE.

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 [4] and Capistrano [2], which provide dashboard-driven workflow-based management of services, and node configuration management tools like Chef [3], and Puppet [6]. The use of workflow to manage service deployments in the cloud however 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 distributed approach to design more robust and scalable management systems. In addition, CLOUDSCAPE is based on the object programming idiom to better structure and extend a system.

Other languages. Other frameworks have been developed to model distributed computation in the cloud, namely Hadoop [1], MapReduce [11], Dryad [14] and Skywriting [15]. 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 do not address the issue of coordination between tasks in a distributed application. Also, the languages of these frameworks describe the control of a system on a central unit, following the workflow approach, avoiding CLOUDSCAPE distributed approach.

VI. CONCLUSIONS AND FUTURE WORK

This paper presented a language to express the dependencies between the lifecycle states of components in a distributed application, and so automatically coordinate and control the components to provide reliability of service when the application is run on the cloud. The three main constructs of the language, namely object, transition and dependency, offer a design on how to build management systems for distributed application. Our language supports the design of coordinating and controlling applications that increase their computing power by adding new components at run-time to provide scalability. Restoring the computation of component due to failure through language design further increases productivity in developing systems for the cloud. A simple, minimal syntax models the three main idioms of the language. The operational semantics rigorously designs the behaviour of objects as autonomous transitions —labeled object— composed in parallel. CLOUDSCAPE provides a semantics that does not allow objects depending on the state of another object to run in parallel while the latter is updating it. We presented a series of examples illustrating the practical utility and effectiveness of this system.

The next step in developing this work is the implementation of the model as a library of SmartFrog. We believe that a library that supports the syntax and semantics of CLOUDSCAPE would increase productivity in the implementation of management systems. From a theoretical perspective, there are several ways to extend the current system. A static type system would allow to statically capture ill-behaved programs and so, guarantee that well-typed programs do not go wrong at run-time. An interesting aspect to further develop is the parallel composition of objects with respect to possible deadlocks, race conditions and starvation when the transition of a component state is defined over more than one attribute. More dynamic concepts such as leaving of components within a session are of interest.

Acknowledgments We thank Brian Monahan for comments on a previous version of this paper.

REFERENCES

[1] "Apache Hadoop," available at http://hadoop.apache.org.
[2] "Capistrano," available at http://www.capify.org/index.php/Capistrano.
[3] "Chef," available at http://www.opscode.com.
[4] "ControlTier," available at http://controltier.org/wiki/Main_Page.
[5] "HP Server Automation," available at https://www.hp.com.
[6] "Puppet," available at http://www.puppetlabs.com.
[7] "SmartFrog," available at http://www.smartfrog.org.
[8] "The Java™ Tutorials: Writing the server side of a socket," available at http://java.sun.com/docs/books/tutorial/networking/sockets/clientServer.html.
[9] M. Abadi and L. Cardelli, *A Theory of Objects*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1996.
[10] D. D. Corkill, "Collaborating and Multi-Agent Systems & the Future," in *International Lisp Conference*, 2003.
[11] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI*, 2004.
[12] K. Fisher, F. Honsell, and J. C. Mitchell, "A lambda calculus of objects and method specialization," *Nordic J. of Computing*, vol. 1, no. 1, pp. 3–37, 1994.
[13] P. Goldsack, P. Murray, M. Newman, and B. Cox, "The Design of a Next Generation Orchestration Engine," in *TechCon*, 2007.
[14] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *EuroSys '07*, 2007, pp. 59–72.
[15] D. G. Murray and S. Hand, "Scripting the cloud with Skywriting," in *HotCloud*, 2010.
[16] A. Silberschatz and J. L. Peterson, *Operating Systems Concepts*. Addison-Wesley, 1988.
[17] V. Subramarian, *Programming Groovy: Dynamic Productivity for the Java Developer*. Pragmatic Bookshelf, 2008.

APPENDIX

This section gives the full definition of the Java `Client` and `Server` classes, following the code of the tutorial [8].

```
import java.io.*;
import java.net.*;

public class Client {
      Socket echoSocket = null;
      PrintWriter out = null;
      BufferedReader in = null;

    public Client(String address, int port) throws IOException {

        try {
            echoSocket = new Socket(address, port);
            out = new PrintWriter(echoSocket.getOutputStream(), true);
            in = new BufferedReader(new InputStreamReader(
                                        echoSocket.getInputStream()));
        } catch (UnknownHostException e) {
            System.err.println("Don't know about host: "+address+".");
            System.exit(1);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for "
                               + "the connection to: "+address+".");
            System.exit(1);
        }
    }
    public void interact(){
        BufferedReader stdIn = new BufferedReader(
                                    new InputStreamReader(System.in));
        String userInput;

        while ((userInput = stdIn.readLine()) != null) {
           out.println(userInput);
           System.out.println("echo: " + in.readLine());
        }
    }
    public void close(){
        out.close();
        in.close();
        stdIn.close();
        echoSocket.close();
    }
}

public class Server {
    ServerSocket serverSocket = null;
    Socket clientSocket = null;
    PrintWriter out = null;
    BufferedReader in = null;

    public Server(int port) throws IOException {
        try {
            serverSocket = new ServerSocket(port);
        } catch (IOException e) {
            System.err.println("Could not listen on port: " +port+".");
```

```
            System.exit(1);
        }
    }
    public void accept(){
        try {
            clientSocket = serverSocket.accept();
        } catch (IOException e) {
            System.err.println("Accept failed.");
            System.exit(1);
        }
    }
    public void interact(){
        out = new PrintWriter(clientSocket.getOutputStream(), true);
        in = new BufferedReader(new InputStreamReader(
                                                clientSocket.getInputStream()));
        String inputLine, outputLine;
        KnockKnockProtocol kkp = new KnockKnockProtocol();

        outputLine = kkp.processInput(null);
        out.println(outputLine);

        while ((inputLine = in.readLine()) != null) {
             outputLine = kkp.processInput(inputLine);
             out.println(outputLine);
             if (outputLine.equals("Bye."))
                break;
        }
    }
    public void close(){
        out.close();
        in.close();
        clientSocket.close();
        serverSocket.close();
    }
}
```

The KnockKnockProtocol class implements the jokes sent to the client. The definition of it can be found in the tutorial [8].

Figure 13 gives the context of expressions that define a local transition.

$$
\begin{array}{rll}
\mathcal{E} ::= & \mathrm{f}\ \mathsf{clone}(L) \leftrightarrow \{\overline{k\ g}\}; g' \\
& \mid N(L', L''); e \\
& \mid \mathsf{local}\ [P]\{e\}; e' \\
& \mid f\ e; e'
\end{array}
$$

Fig. 13. Evaluation Context