Modeling and Testing of Cloud Applications*

W.K. Chan†
City University of Hong Kong
Tat Chee Avenue, Hong Kong
wkchan@cs.cityu.edu.hk

Lijun Mei
The University of Hong Kong
Pokfulam, Hong Kong
ljmei@cs.hku.hk

Zhenyu Zhang
The University of Hong Kong
Pokfulam, Hong Kong
zyzhang@cs.hku.hk

***Abstract*—What is a cloud application precisely? In this paper, we formulate a computing cloud as a kind of graph, a computing resource such as services or intellectual property access rights as an attribute of a graph node, and the use of a resource as a predicate on an edge of the graph. We also propose to model cloud computation semantically as a set of paths in a subgraph of the cloud such that every edge contains a predicate that is evaluated to be true. Finally, we present algorithms to compose cloud computations and a family of model-based testing criteria to support the testing of cloud applications.**

Keywords—cloud application; graph; testing criteria

I. INTRODUCTION

Cloud computing [8][16] is an emerging trend to deploy and maintain software and is being adopted by the industry such as Google [14], IBM [7], Microsoft [26], and Amazon [1]. Several prototype applications and platforms, such as the IBM "Blue Cloud" infrastructure [7], the Google App Engine [15], the Amazon Cloud [1], and the Elastic Computing Platform [11], have been proposed. However, when it comes to the question on how to model cloud applications (e.g., [9][30]), the question remains unexplored. In our previous work [21][25], we put forward several issues toward developing cloud applications. In this paper, we sketch an application model, and develop theoretical test adequacy criteria for testing applications in a cloud.

There was a debate on programming-in-the-large versus programming-in-the-small [10]. It led to the consensus in the software engineering community that software methodologies and techniques to support the former kind can be different from those for the latter kind. On the other hands, we observe that many recent proposals on cloud computing are "in the large", such as focusing on scaling an application to the internet scale transparently or without much user intervention. There is little discussion on the "in the small" side. We incline to believe that such a "small cloud" could be more manageable than a huge cloud, and thus having a more uniform strategy to reason or manage cloud applications may be viable.

In this paper, we present a semantic model to support modeling, analysis and testing of computing "clouds *in-the-small*". We first formulate the notion of a bare-bone cloud as a foundation for modeling and analyzing cloud computing. We use selected features of the real-life weather cloud system as a metaphor to refine the notion of bare-bone clouds to a kind of directed graph, which we call a *cloud graph*. In a cloud graph, every node is a computing entity. A computing resource such as a service or an intellectual property (IP) access right to use a particular service or data (e.g., image or photo) is modeled as an attribute of a node. The availability of an attribute of one node to another node is modeled as a predicate on an edge that connects from the latter node to the former one. Thus, a cloud execution can be semantically modeled as a set of paths in a *predicate-enabled* subgraph of a cloud graph.

We also develop algorithms to manipulate cloud computations. Furthermore, we propose theoretical test adequacy criteria to assure the quality of such cloud applications. Although our model may be applicable to clouds of different scales, our algorithms are particularly viable to clouds *in-the-small*, in the sense that a process (in the system sense) is capable to oversee the activities of the cloud and exercise cloud management.

The main contribution of this paper is threefold. (*i*) We present a graph-theoretic model of computing clouds. (*ii*) We formulate how to transform, compose, and decompose cloud graphs, in which the cloud computations are taking place. (*iii*) We propose the first set of model-based testing criteria for testing cloud applications.

The rest of the paper is organized as follows: Section II presents the concept of bare-bone clouds. Section III uses a metaphor to show three characteristics of a weather cloud system, and maps these characteristics to the properties of computing clouds. Section IV presents a cloud graph model, discusses its properties and behaviors, and develops a family of testing criteria, followed by a literature review in Section V. Section VI concludes the paper.

II. BARE-BONE CLOUDS

In this section, we present a bare-bone model to facilitate software designers to reason the composition and decomposition of computing clouds to meet the requirements of their applications. This model will also be used as the basis to derive our cloud graph model (in Section IV).

In our bare-bone model, a computing cloud is modeled as a directed graph *c*, representing a grid of computing resources.

* This research is supported in part by the General Research Fund of the Research Grant Council of Hong Kong (project nos. 123207 and 717308), and the Strategic Research Grant of City University of Hong Kong (project no 7002464).

† Correspondence author.

978-1-4244-5336-8/09/$26.00 ©2009 IEEE

Each computing resource can be a service [3][4], IP rights, computing power, persistent storage, memory, or network bandwidth (that connects multiple computing resources). We model such a bare-bone cloud c as a graph $\langle V, E\rangle$, where V is a set of nodes, denoting the providers of computing resources, and E ($\subseteq V \times V$) is a set of edges, and each edge relates to two providers that communicate directly with each other at the application level. Because different providers may offer different kinds of computing resources, each node n ($\in V$) is also associated with a set of computing resources $\{r_1, r_2, \ldots, r_k\}$. A *subcloud* is a connected subgraph of a cloud. The same resource r_i may exist at multiple nodes in the same cloud, or may have been "virtualized" [13].

We refer to a client that uses a computing resource as a *cloud consumer* (or simply a *consumer*), which is also a node in the cloud. For instance, a Hong Kong-based parcel agency may develop a tailor-made service that directly communicates with the Google Map web service so that a consumer can use Google Map with mash up to locate their parcels.

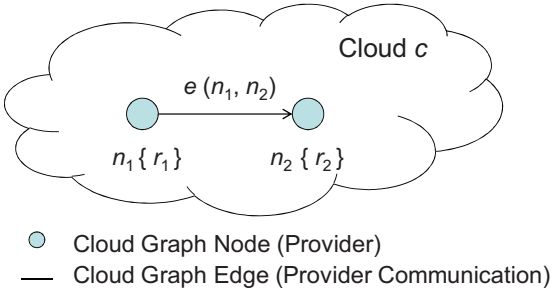


Figure 1. Example of bare-bone cloud.

In the above scenario, the location of a particular Google Map service is transparent to the parcel agent. For the ease of presentation, we refer to the computing cloud as c. In the bare-bone model, the Google Map provider is represented by a node n_1 in the graph c, and n_1 is associated with a map service r_1. We use the notation $n_1.r_1$ to denote the consumption of the computing resource r_1 available at node n_1. Similarly, the parcel agency can be modeled as a node (say, n_2) that is associated with the tailor-made service r_2. There is also an edge $\langle n_1, n_2\rangle$ in the cloud c to denote the consumption of a service in a cloud, which is shown as a relation among n_1, n_2, r_1, and r_2 in the cloud. This edge is illustrated in Figure 1.

III. A Metaphor for Cloud Computing

In this section, we study the lifecycle of a real-life weather cloud as a metaphor to enrich our model.

A. Weather Cloud as a Metaphor

We observe that a weather cloud exhibits at least three characteristics.

(C1) The shape of a weather cloud changes constantly. Moreover, the entropy of the cloud and environmental factors such as pressure and wind play important roles in changing the shape of the cloud. Furthermore, there is usually a *chain reaction*, rather than a single action-reaction pattern. On the other hand, such a cloud reacts *passively* to these environmental factors.

(C2) The water vapor grains that constitute a cloud may vary in size, type, shape, and composition. Different grains may merge to become a bigger grain, or a grain may decompose into smaller grains. However, once a composition or decomposition of grains has started, it is impractical to reverse the process. For this reason, the original state of a cloud can be too costly to restore. This observation leads us to *obsolete the notion of keeping the history* of a cloud in our cloud model.

(C3) Multiple clouds may merge to become a united cloud. Unlike object aggregation in the sense of object-oriented modeling, the original composing elements of this newly formed cloud can hardly be distinguished. This observation leads us to *obsolete the notion of keeping the boundaries of sub-clouds* in our cloud model.

B. Cloud Computing Based on the Metaphor

Following the highlighted characteristics (C1–C3) of the metaphor in the last section, we proceed to study the mapped characteristics in cloud computing.

(M1) Computing clouds should be adaptive. Whenever a computing cloud detects changes in its environment, it needs to adjust itself to the new situation. Furthermore, according to our observation on chain reaction in C1, a cloud evolution is likely to trigger new changes in the environment, and hence the cloud will evolve further. In general, there is no explicit equilibrium point for such evolutions because the cloud is an open system, and there are frequent changes in the environment.

(M2) Computing clouds should only marginally depend on the history. When a cloud is composed from subclouds, every individual subcloud may involve different types and quantities of computing resources. According to our observation on the forgotten history in C2 and the passive reactions to the environmental changes in C1, the functionality of the composed cloud should be strongly decoupled from the historical events.

(M3) Computing clouds are typically tightly-coupled. When the computing resources in a cloud cannot satisfy a computing requirement (such as processing a transaction to store a huge file in the network), the cloud can be merged with another one to seek additional such resources. The extent of cloud integration may, however, vary. For instance, if the integration merely seeks sharing of certain resources, a way is to link up multiple clouds. Clouds in such a bridged cloud cluster can be loosely coupled. Nevertheless, after cloud integration, the computing resources may need to be redistributed among clouds in the cluster. These clouds then become tightly coupled, and any split of the cloud may affect the computations taking place.

Thus, adding such a bridge will result in chain reactions (see C1) within a cloud cluster, which is then transformed into a set of tightly coupled clouds. As such, a cloud cluster is

hardly separable, and the assumption of a loosely coupled one appears to be out of the norm. Thus, keeping the boundaries of subclouds serves little practical purpose and may only increase the complexity of cloud management, which is of course undesirable (see C3).

IV. Modeling and Testing Cloud Computation

In this section, we present a model to formulate computing clouds in-the-small.

A. Formulation

We propose to model the environment as a cloud as well. It simplifies the model so that an interaction between the environment and a cloud can be modeled as an interaction between two clouds [32] (dubbed as a *cloud interaction*). Thus, a chain reaction, possibly with the environment, can be modeled as a sequence (or directed graph) of cloud interactions.

In our bare-bone model (see Section II), a cloud is a directed graph of providers and consumers. Each provider carries a set of computing resources. However, the access of resources has not been modeled. Thus, we extend a cloud with a set (possibly empty) of labels attached to the edges of the cloud graph. Each of these labels is a predicate over the set of computing resources in the cloud. Such a predicate decides whether the providers (that is, the nodes associated with the edge) have the computing resources available for consumption through the edge.

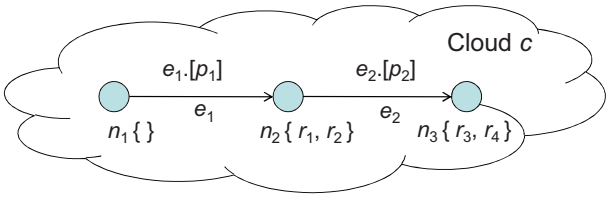
Definition 1 (Cloud Graph). A *cloud graph* is a 4-tuple $G\langle V, E, P, R\rangle$. $\langle V, E\rangle$ is a bar-bone cloud. Every node $v \in V$ is associated with a resource set $\{r_1, r_2, \ldots, r_n\}$, where each $r_i \in R$ is some computing resource. Every edge $e \in E$ is associated with a predicate set $\{p_1, p_2, \ldots, p_m\}$, where each $p_i \in P$ is a first-order predicate over the computing resource variables.

We also use the notation $e.[p]$ to denote the predicate p on the edge e. We say that the binding of variables in the predicate p is *well formed* if every variable is successfully bound to the computing resources of the nodes associated with the edge e. In other words, for every variable x on $e.[p]$, if $e = \langle n_1, n_2\rangle$, then x should be bound to a resource in either n_1 or n_2. We further impose a health constraint on our model: *only those well-formed predicates can be evaluated to be true or false*.

If an edge has a predicate that has been evaluated to be true, then the edge is said to be *enabled*. Otherwise, it is said to be *disabled*. Since an edge in the bare-bone model represents a direct communication between two providers, an enabled edge thus indicates that the underlying computing resources support the communication between the providers. A disabled edge models a potential (but inactive) communication between a consumer and a provider.

In our model, edge enabling is an important concept to support the reasoning of cloud computation. For instance, a primary cloud consumer may use a resource provided by a primary cloud provider, which, in turn, acts as a secondary cloud consumer that requires other computing resources from other secondary providers, and so on. This scenario can be modeled by a sequence of enabled edges in a cloud graph.

Formally, an *enabled subcloud sc* is a subgraph of a cloud c such that every edge is enabled. However, not every enabled subcloud represents a cloud computation. Consider Figure 2, where two edges $e_1 = \langle n_1, n_2\rangle$ and $e_2 = \langle n_2, n_3\rangle$ connect two nodes n_1 and n_3 via a third node n_2. Suppose n_2 has two resources r_1 and r_2. The predicate p_1 on edge e_1 is well formed by successfully binding variable x to r_1. The predicate p_2 on e_2 is well-formed by binding variable y to r_3 or r_4. In this way, e_1 can be enabled when r_1 is available, and e_2 can be enabled when either r_3 or r_4 is available.



p_1 : *bound* (r_1, x) → *enable* (e_1)

p_2 : *bound* (r_3, y) ∨ *bound* (r_4, y) → *enable* (e_2)

x, y : variables

r_1 - r_4 : resources

bound is true when a resource is bound to a variable; otherwise, false.

enable sets the enabling of an edge true.

Cloud Graph Node (Provider)

Cloud Graph Edge (Provider Communication)

Figure 2. Example of enabled subcloud.

In this example, the consecutive edges are *not* connected via *shared* computing resources. Thus, we add two more health constraints to our model that represents cloud computation (see Definition 2.)

Definition 2 (Cloud Computation). A *cloud computation* Ω for a cloud consumer n of a cloud c is a set of paths in an enabled subcloud c' of c satisfying two conditions: (i) n is a node of c'. (ii) For any path $\rho \in \Omega$ and for any two consecutive edges (say e_1 and e_2) on ρ, the node (say n') connecting e_1 and e_2 should have at least one computing resource bound to the same set of variables that simultaneously enable at least one predicate on each of e_1 and e_2.

B. Properties of cloud graphs

In this section, we, referencing the graph theory [31], define a few utility properties of the cloud graph. They will be used in the next section.

Definition 3 (Cloud Computation Distance). A *cloud graph distance* for a cloud computation Ω in a cloud graph c, denoted by $Dist(\Omega)$, is the length of the shortest computation path in Ω.

Obviously, $Dist(\Omega) = Dist(\{\rho\})$ if and only ($\rho \in \Omega$ and, $\forall \rho' \in \Omega$, $Dist(\{\rho\}) \le Dist(\{\rho'\})$).

The cloud computation distance measures the length of the shortest invocation sequence in a cloud graph. Due to the changing connectivity of the cloud graph, our heuristics is that the smaller the cloud computation distance is more stable (with respect to changes of structure to meet the change in the cloud as a whole) the computation will be. Furthermore, if all edges have the same cost, a lower cost is expected with a smaller cloud computation distance. One may use the Dijkstra's shortest path algorithm to find such a distance.

However, we note that different edges in a cloud graph c may represent different distances and qualities. Therefore, we further propose a *weighted cloud computation distance* to distinguish such cases. A *weighted cloud computation distance* for a cloud computation Ω in a cloud graph c is dubbed as *WeightDist* (Ω), which calculates the weighted length of a cloud computation in c. One may use a weighted version of the Dijkstra's shortest path algorithm to find such a distance.

Next, we define *cloud graph connectivity* that aims to reveal the internal structure of a cloud graph. The connectivity will also be used as the base of merging and splitting a cloud graph. We refer to the graph theory [31] to define *edge-connectivity* of a cloud graph.

Definition 4 (Cloud Graph Connectivity). A cloud graph G is said to be k-connected (k-edge-connected) if its edge connectivity is k or more. The edge connectivity is the size of a smallest edge cut. An *edge cut* of G is a set of edges whose removal renders G disconnected.

We note that there are many algorithms to find edge cuts in a given graph, and we denote such an algorithm as *find-edgecut-set*(G). In the next section, we will use these properties to develop the algorithms to model cloud computations.

C. Cloud graph interaction

Based on the definitions in Section IV(B), we proceed to model a cloud interaction between two clouds. As mentioned in Section III, a cloud interaction represents a situation that a cloud may *grow* or *shrink*. Rather than studying a passive cloud, we study how a cloud computation can be grown or shrunk actively.

A cloud interaction may be feasible if it happens between the enabled subclouds of two clouds; or else, it lacks in computing resources to enable the interactions. (Due to space limit, we omit the proof.) We further observe that a computation should take place during a cloud interaction; otherwise, there is no enabled subcloud in at least one cloud, prohibiting a cloud interaction from occurring. Based on such observations, we refine the idea of cloud interaction to the interaction of cloud computation.

Definition 5 (Interaction of Cloud Computation). Given two cloud computations Ω_1 and Ω_2, if there are common sub-paths between Ω_1 and Ω_2 (i.e., $\Omega_1 \cap \Omega_2 \neq \emptyset$), we say that there is an interaction between Ω_1 and Ω_2. (Note that, we have overloaded the symbols Ω_1 and Ω_2 to refer to their path sets, respectively.)

Based on this interaction concept, we further identify that the computing resource binding for a common sub-path on Ω_1 may or may not be the same as that on Ω_2. Let us consider two scenarios to illustrate our model. (However, our model is not just applicable to these two scenarios.)

Inconsistency detection on cloud graph. First, if these two cloud computations compete for a shared computing resource on a node, it will result in resource contention. This can be checked via a subsequence checking operation between paths of Ω_1 and Ω_2 to identify whether there is any common sub-path (i.e., $\Omega_1 \cap \Omega_2 \neq \emptyset$). Once such a common sub-path has been identified, the predicates of both Ω_1 and Ω_2 on the sub-path can be further checked on whether these predicates use the same computing resources of nodes on the sub-path. If so, a resource confliction is detected.

Cloud partitioning. Second, given a set of cloud computations, we can determine whether two cloud computations may share any edges or nodes. If so, we may merge these two cloud computations to become one cloud computation. We can repeat the merging process until no two clouds share any edges or nodes. Thus, the original computing cloud is readily partitioned into multiple subclouds, each subcloud containing a cloud computation, and a (remaining) cloud consisting of nodes that are not involved in any cloud computations (dubbed as a *buffer cloud*). Individual subclouds can then be used for further analysis or optimization. When a cloud computation is completed, the belonging subcloud may merge with the buffer cloud. However, the procedures to split a cloud by using the current and local state still require more research.

D. Dynamic cloud graph composition

In this section, we demonstrate how our cloud graph model can be used in the dynamic composition of computing cloud.

Since not all clouds are of the same type, these clouds cannot simply merge into the same type of cloud. Therefore, from this point of view, the computing clouds can be considered to be provisioned for different tasks. When the task requirement changes, or the computing resources change, a cloud may require modifying its embedding cloud computation to optimize the fulfillment of this task.

The changes indicate a need of an algorithm to reform the original cloud graph to a new cloud graph so that the existing cloud computations may continue to execute. Similarly, a cloud may split into multiple clouds, or multiple clouds may merge together. To support such scenarios, we propose three algorithms.

1) Cloud graph reform procedure.

We present the algorithm *Reform_CloudGraph* to reorganize a cloud computation so that it adapts to the changes in resource binding.

Algorithm *Reform_CloudGraph*

Inputs Cloud graph $c\langle V, E, P, R\rangle$
Outputs Cloud graph $c\langle V, E, P, R\rangle$

1 **for each** cloud computation Ω in c **do**
  // Find the shortest cloud computation path:
2 $\Omega_\rho \leftarrow \{\rho \mid \forall \rho, \rho' \in \Omega, Dist(\{\rho'\}) \geq Dist(\{\rho\})\}$
3 **for each** $\rho \in \Omega$ **do**
4 **if** $\rho \in \Omega_\rho$ **then**
5 *EnablePath*(ρ)
6 **else**
7 *DisablePath*(ρ)
8 **end if**
9 **end for**
10 **end for**
  // Check if potential cloud computation path exists:
11 **for each** $\rho \in \Omega$ **do**
12 **if** $\exists n_i, n_j, n_k \in V, \langle n_i, n_j\rangle \in E, \langle n_i, n_k\rangle \notin E$, **then**
13 let R_j and R_k be the resource set of n_j and n_k
14 let R_j' ($\subseteq R_j$) be the resource subset that n_i consumes from R_j
15 **if** $R_j' \subseteq R_k$ **and** *EdgeDist* (n_i, n_k) < *EdgeDist*(n_i, n_j), **then**
  // Add a new edge to the graph:
16 $e \leftarrow \langle n_i, n_k \rangle$
17 let p be the predicate formed by R_j' for e
18 $e.[p] \leftarrow$ true // i.e., enable the edge
19 $E' \leftarrow E \cup \{e\}$
20 **end if**
21 **end if**
22 **end for**
23 **if** the input cloud graph ≠ the output cloud graph **then**
24 **return** *Reform_CloudGraph*(c)
25 **else**
26 **return** c
27 **end if**

In this algorithm, we define the functions *EnablePath* and *DisablePath* to bind and unbind resources on each edge of a computation path, and define the function *EdgeDist* to calculate the distance of a single edge, which can represent network latency, cost, or other measures on the QoS attributes of the edge. The algorithm accepts a cloud graph, and iteratively removes those edges that are not used by the current computation (#1–#10). It then looks for alternative resources provider (n_k) of the current resources provider (n_i), and replaces the latter node by the former node if the former one is closer to the original node than the latter one. Thus, the algorithm uses a hill-climbing strategy to optimize the overall edge distances of each cloud computation.

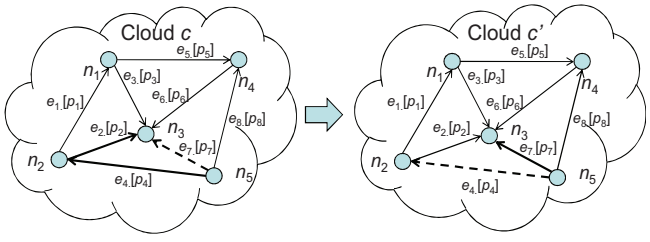


Figure 3. An example cloud graph *reform*.

An example showing the *reform* of a cloud graph c is given in Figure 3. Suppose the edge $\langle n_5, n_3\rangle$ in the cloud graph c (dashed line) has a smaller distance than through the edges $\langle n_5, n_2\rangle$ and $\langle n_2, n_3\rangle$. Therefore, we transform cloud graph c to c' through enabling edge $\langle n_5, n_3\rangle$ and disabling edge $\langle n_5, n_2\rangle$.

The cloud graph reform procedure can be invoked right after the cloud graph changes. Two basic operations on changing a cloud graph are graph splitting and graph merging.

2) Cloud graph splitting procedure.

A cloud graph c can be split into multiple subgraphs $\{c_1, c_2, \ldots, c_n\}$ if the cloud graph connectivity (see Definition 4) is not more than a defined ceiling. Intuitively, the ceiling parameter controls the strength of coupling among nodes within each cloud computations (as captured by a notion of cloud).

We present an algorithm *Split_CloudGraph* to show how a cloud graph can be split into multiple subgraphs.

Algorithm *Split_CloudGraph*

Inputs Cloud graphs c
Outputs Subcloud graphs $c_1, c_2, \ldots, c_n$
  // Calculate the connectivity of cloud graph c:
1 $k \leftarrow$ *connectivity* (c)
  // Split cloud graph c if $k \leq$ *SPLIT_LEVEL*:
2 **if** $k \leq$ *SPLIT_LEVEL* **then**
  // Find a set of edge cut of cloud graph c
3 $E' \leftarrow$ *find-edgecut-set* (c)
  // Remove edges in the edge cut sets:
4 $E \leftarrow E \setminus E'$
5 **end if**
6 collect the disconnected subgraphs as C
  // Recursively process the disconnected subgraphs:
7 **for each** $c' \in C$ **do**
8 *Split_CloudGraph*(c')
9 **end for**

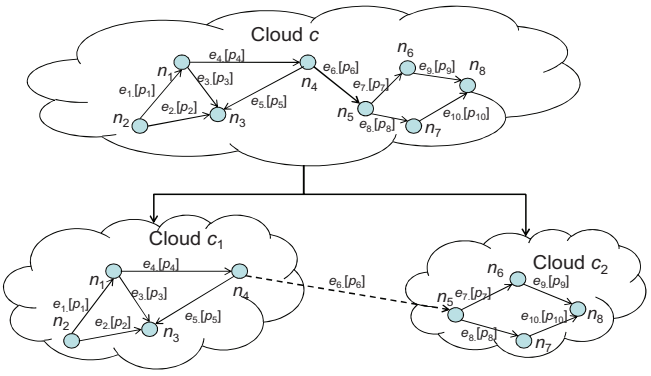


Figure 4. An example cloud graph *split*.

Each subgraph represents a sub-computation if it contains at least one node that at least one of its connecting edges has been enabled. In some cases, a split cloud may contain "idle" nodes

and edges, and thus, this sub-graph essentially represents no computation, and can be removed from the cloud computation set.

An example showing the split of cloud graph c is given in Figure 4. Suppose the *SPLIT_LEVEL* is set to be 1. Cloud graph c can be split into two subgraphs c_1 and c_2. The cloud graph split procedure can be invoked automatically or manually.

3) Cloud graph merging procedure.

We present an algorithm to show how two cloud graphs can be merged.

Algorithm *Merge_CloudGraph*

Inputs Cloud graphs $c_1\langle V_1, E_1\rangle$, $c_2\langle V_2, E_2\rangle$
Outputs (Merged) cloud graph $c\langle V, E\rangle$

```
// Collect the interactions between c1 and c2:
1  V ← ∅, E ← ∅
2  numOfInteraction ← 0
3  for each v1∈ V1 do
4      for each v2∈ V2 do
5          if ∃e = ⟨v1, v2⟩ or ⟨v2, v1⟩ such that e.P is true
6          then
             numOfInteraction ← numOfInteraction + 1
7          end if
8      end for
9  end for
// Merge cloud graphs if the number of interactions is
// above MERGE_LEVEL:
10 if numOfInteraction ≥ MERGE_LEVEL then
       // First, combine cloud graphs c1 and c2 into c:
11     V ← V1 ∪ V2
12     E ← E1 ∪ E2
13 end if
   // Then, reform c:
14 Reform_Graph(c)
```

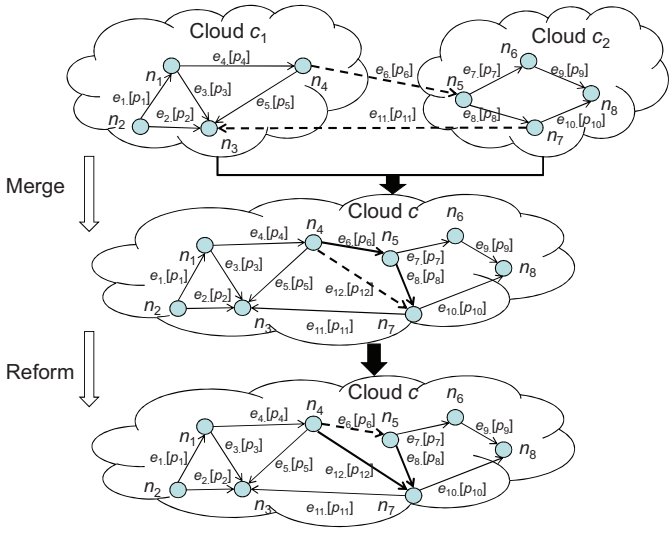


Figure 5. An example cloud graph *merge*.

More than two cloud graphs can also be merged using this algorithm iteratively. An example showing the merging of cloud graph c is shown in Figure 5. Suppose the *MERGE_LEVEL* is set to be 2. Two cloud graphs c_1 and c_2 have two interactions (shown in dashed lines). Then we form a union of c_1 and c_2 into cloud graph c. After that, we reform c.

Contrast to the cloud graph splitting procedure, the cloud graph merging procedure can be invoked when certain thresholds of cloud clusters have been reached. When graphs are merged, there will be opportunities to share resources that are not feasible because the related resources may be located in disconnected cloud graphs. Therefore, for optimization purpose, the algorithm *Reform_CloudGraph* can optionally be invoked right afterward.

We formulate the notion of self-optimization (*reform*) of a cloud to address both evolving resource qualities and the changing environment. In particular, we use a hierarchical and incremental approach to *merge* or *split* cloud graphs. Suppose, for instance, that a mobile device has been modeled as a cloud graph c consisting of one node. When the device moves to another location, it usually needs to disconnect from the current cloud (say, c_1) and connect to another one (say, c_2). Such a procedure happens frequently to mobile devices. We can represent such actions through the *split* and *merge* procedures. Moreover, the environmental data of cloud c can be transferred from its previous surrounding cloud c_1 to current surrounding cloud c_2.

E. Testing

Testing is the de facto activity to assure the quality of an application. We believe that cloud application is not an exception. In general, testing criteria define whether adequacy test has been conducted. To the best of our knowledge, there is no proposal in the literature on testing criteria [20][23] for assuring the quality of cloud applications. This section proposes a couple of such testing criteria.

The first criterion (*all-predicates*) tests whether the application has decided to use the resources properly. If safety is a requirement, this criterion can be further refined into a family of well-known MC/DC-like testing criteria. Owing to page limit, we omit this family in this paper.

Criterion 1 (***all-predicates***): Given a cloud computation graph c, the *all-predicates* criterion is fulfilled by a test set T if every predicate in c has been exercised by at least once test case in T.

The second criterion is to test whether the application can be performed correctly after horizontal scaling of the cloud. However, there are potentially infinite numbers of possible scaling. Thus, it is infeasible to test every configuration. We resolve to test whether computational equivalence [29] can be achieved after mutation of the cloud graph. Such mutation can be achieved through simulation and virtualization techniques.

We define that a cloud graph m is called a mutant of a cloud graph c if (1) one of the predicates of c has been mutated using a mutation operator in the sense of mutation testing [33]

to form *m*, (2) one of the nodes or edges has been removed from *c*, or (3) one of the nodes of *c* has been duplicated in *d* and thus relabeled to a new distinct node of *d*. A mutant is said to be killed if the output of the mutant is not the same as that of the original program. (We note that in our model, the output can be measured at the predicate level or node level.)

Criterion 2 (***all-reforms***): The all-reforms criterion is said to be fulfilled by a test set *T* if every mutant (after applying the algorithm *Reform_CloudGraph*) can be killed by *T*.

Since cloud scalability should be transparent to a cloud computation, thus even though mutation has been occurred, the computation should not be affected if it can compute an output. Chances are, the mutants will make certain resources (via predicate mutants) unavailable for the cloud computation under test, and lead the applications to produce non-equivalent results or the "execution" simply crashes. This property forms a correctness criterion to test and analyze cloud computation in our model.

Owing to page limit, testing criteria to test applications against cloud splitting and merging have not been presented. We also note that the above-mentioned testing criteria are theoretical in nature. We are studying whether they can be effective by examining the fault classes [17] that have been developed in the software engineering community. We have not studied the effectiveness and the feasibility of such testing criteria in details. The way to define a test case should also require more studies.

V. Related Work

This section reviews the literature related to our work.

The paradigm of grid computing is close to that of cloud computing. Foster and Kesselman [12] take the grid as a computing infrastructure and introduce the notion of grid computing. They illustrate how grids can be used to solve research problems such as diagnostic problems and the Aero-engine DP problem. Existing research (e.g., [5][6][13]) on grid computing focuses on the computing resource organization and computing task distribution. On the contrary, cloud computing emphasizes user experience when using cloud services. The availability of virtualized resources becomes a key factor. Our model explicitly incorporates resources as a key dimension.

Next, we review the context-aware computing. Context-aware computing is important to provide adaptive behaviors to systems. Lu et al. [20] propose a technique to test pervasive software surrounded by different services. Mokhtar et al. [27] illustrate the problem of composition in the environment of pervasive computing. Lee et al. [18] propose to use a smart space middleware to hide the complexity involved in context-aware and automated service composition. Anhalt et al. [2] outline a general solution to support contextual awareness. Our previous work [21][25] discusses the context-awareness of cloud computing by comparing the key characteristics of cloud computing with pervasive computing and services computing [28]. Our model has put special focus on modeling the environmental contexts of clouds. It is because each computing device in a cloud can be deployed on different machines, the environmental contexts may play an important role in determining the quality of the resultant clouds.

Compared to the service-oriented applications, many researchers have suggested that a computing cloud may also provide services. Our previous work [21][25] compares the key characteristics of cloud computing and services computing. Lin et al. [19] put cloud computing and IT as a Service (ITaaS) together, and propose to study them from both the technology and business model perspectives. Our previous work [22] proposes to solve the service selection problem by using link analysis techniques. In cloud computing, different computing resources also need to be evaluated and ranked. As such, only qualified resources will be used by the computing clouds. Such filtering process will increase the quality of the computing clouds. Testing criteria for service-related systems have been proposed (e.g., [20][23][24]), but we are not aware of any existing testing criteria for cloud applications.

Finally, cloud interactions can be considered similar to the interactions among services. However, we have learnt from services computing that such consumption or data exchange between services may result in integration problem that may affect cloud compositions. Assuring the quality and providing dependability of cloud interactions warrant more research efforts.

VI. Concluding Remarks

Cloud computing is an emerging computing model that requires more research attention. In this paper, we have presented a graph-theoretic model aiming to describe and reason applications of computing cloud in the small and their interactions. We have studied the concept of a cloud as a graph, the representation of resources as node attributes, the use of resources as predicates, and an execution as a set of directed paths of a cloud graph. Our model can be viewed as a kind of predicate-based graph.

Through the notion of predicate-enabled subclouds, we have studied how cloud interactions can be captured and represented by our model to support formal analysis. We have further illustrated how to use our model to conduct analysis on cloud composition and detection of anomalies. We have further proposed model-based test adequacy criteria to support the testing of cloud applications.

Our model also has several limitations. Currently, it only supports stateless atomic operations or cloud computations that can be expressed in the form of context-free grammars. One may incorporate different types of scalability, exception handling, and dynamic binding among attributes of nodes. Service transactions and explicit concurrency have not been studied. Model development to address them could be valuable.

Acknowledgment

We thank Prof. T.H. Tse of The University of Hong Kong for his discussion on an earlier version of this paper. We also thank the anonymous reviewers for their helpful comments.

REFERENCES

[1] *Amazon Elastic Compute Cloud*. Available at http://aws.amazon.com/ec2/. (Last access September 12, 2009.)

[2] J. Anhalt, A. Smailagic, D. P. Siewiorek, F. Gemperle, D. Salber, S. Weber, J. Beck, and J. Jennings. Toward context-aware computing: experiences and lessons. *IEEE Intelligent Systems*, 16 (3): 38–46, 2001.

[3] B. Benatallah, R. M. Dijkman, M. Dumas, and Z. Maamer. Service-composition: concepts, techniques, tools and trends. In *Service-Oriented Software System Engineering: Challenges and Practices*, pages 48–66. Idea Group, Hershey, PA, 2005.

[4] M. Broy, I. H. Kruger, and M. Meisinger. A formal model of services. *ACM Transactions on Software Engineering and Methodology* (*TOSEM*), 16 (1): Article No. 5, 2007.

[5] R. Buyya. Economic-based Distributed Resource Management and Scheduling for Grid Computing. PhD Thesis. Monash University, Melbourne, Australia, 2002.

[6] R. Buyya, C. S. Yeo, and S. Venugopal. Market-oriented cloud computing: vision, hype, and reality for delivering IT services as computing utilities. In *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications* (*HPCC 2008*), pages 5–13, 2008.

[7] *Cloud Computing*. IBM. Available at http://www.ibm.com/ibm/ cloud/. (Last access September 12, 2009.)

[8] *Cloud Computing*. Wikipdia. Available at http://en.wikipedia.org/wiki/Cloud_computing. (Last access September 12, 2009.)

[9] K. A. Delic and M. A. Walker. Emergence of the academic computing clouds. *Ubiquity*, 9 (31): 1, 2008.

[10] F. DeRemer and H. Kron. Programming-in-the large versus programming-in-the-small. In *Proceedings of the international Conference on Reliable Software*, pages 114–121, 1975.

[11] Enomaly's Elastic Computing Platform in Sourceforge. Available at http://sourceforge.net/projects/enomalism/. (Last access September 12, 2009.)

[12] I. Foster and C. Kesselman (editors). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, San Francisco, CA, 1999.

[13] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15 (3):200–222, 2001.

[14] Google and the wisdom of clouds. *Business Week*. 2007. Available at http://www.businessweek.com/magazine/content/07_52/b4064048925836.htm.

[15] *Google App Engine*. Google. Available at http://code.google.com/appengine/. (Last access September 12, 2009.)

[16] B. Hayes. Cloud computing. *Communications of the ACM* (*CACM*), 51 (7): 9–11, 2008.

[17] M. F. Lau and Y. T. Yu. An extended fault class hierarchy for specification-based testing. *TOSEM*, 13 (3):247–276, 2005.

[18] C. Lee, S. Ko, S. Lee, W. Lee, and S. Helal. Context-aware service composition for mobile network environments. In *Ubiquitous Intelligence and Computing*, volume 4611 of Lecture Notes in Computer Science, pages 941–952, 2007.

[19] G. Lin, G. Dasmalchi, and J. Zhu. Cloud computing and IT as a service: opportunities and challenges. In *Proceedings of the IEEE International Conference on Web Services* (*ICWS 2008*), page 5, 2008.

[20] H. Lu, W. K. Chan, and T. H. Tse. Testing pervasive software in the presence of context inconsistency resolution services. In *Proceedings of the 30th International Conference on Software Engineering* (*ICSE 2008*), pages 61–70, 2008.

[21] L. Mei, W. K. Chan, and T. H. Tse. A tale of clouds: paradigm comparisons and some thoughts on research issues. In *Proceedings of the 2008 IEEE Asia-Pacific Services Computing Conference* (*APSCC 2008*), pages 464–469, 2008.

[22] L. Mei, W. K. Chan, and T. H. Tse. An adaptive service selection approach to service composition. In *Proceedings of ICWS 2008*, pages 70–77, 2008.

[23] L. Mei, W. K. Chan, and T. H. Tse. Data flow testing of service-oriented workflow applications. In *Proceedings of ICSE 2008*, pages 371–380, 2008.

[24] L. Mei, W. K. Chan, and T.H. Tse. Data flow testing of service choreography. In *Proceedings of ESEC/FSE 2009*, pages 151-160, 2009.

[25] L. Mei, Z. Zhang, and W.K. Chan. More tales of clouds: software engineering research issues from the cloud application perspective. In *Proceedings of COMPSAC 2009*, pages 525–530, 2009.

[26] Microsoft plans 'cloud' operating system. *New York Times*. 2008. Available at http://www.nytimes.com/2008/10/28/technology/28soft.html.

[27] S. B. Mokhtar, D. Fournier, N. Georgantas, and V. Issarny. Context-aware service composition in pervasive computing environments. In *Rapid Integration of Software Engineering Techniques*, volume 3943 of Lecture Notes in Computer Science, pages 129–144, 2006.

[28] D. Saha and A. Mukherjee. Pervasive computing: a paradigm for the 21st century. *IEEE Computer*, 36 (3): 25–31, 2003.

[29] T. H. Tse, S. S. Yau, W. K. Chan, H. Lu, and T. Y. Chen. Testing context-sensitive middleware-based software applications. In *Proceedings of the 28th Annual International Computer Software and Applications Conference* (*COMPSAC 2004*), volume 1, pages 458–465, 2004.

[30] A. Weiss. Computing in the clouds. *netWorker*, 11 (4):16–25, 2007.

[31] D. B. West. *Introduction to Graph Theory*, Prentice Hall, 2nd Edition, Chapter 2 and Chapter 4, 2001.

[32] When clouds collide. *The Economist*. 2008. Available at http://www.economist.com/business/displaystory.cfm?story_id=10650607.

[33] Wikipedia. Mutation testing. Available at http://en.wikipedia.org/wiki/Mutation_testing.