Synthesis of Cloud Applications using Logic Programming: BAM!

Ethan K. Jackson and Wolfram Schulte
Research in Software Engineering,
Microsoft Research, Redmond, WA, USA
{ejackson, schulte}@microsoft.com

Daniel Lucrdio
Institute of Mathematical and Computer Sciences,
University of São Paulo, São Paulo, Brazil
lucredio@icmc.usp.br

Abstract

Cloud applications are web-based distributed systems deployed over a fluctuating set of computing nodes and services. The design of cloud applications is particularly challenging because few assumptions can be made about the connectivity of nodes, the availability of services, as well as how the computing fabric will evolve in the long term. In this paper we show that logic programming combined with novel abstractions can be used to specify and synthesize cloud applications. Our tool-suite, called BAM, allows cloud applications to be specified independently from implementation technologies. The declarative nature of logic programming is essential to produce this decoupling. Code synthesis integrates state-of-the-art web technologies with well-known algorithms from logic programming to produce realistic and complete implementations from BAM specifications. First experiments show that mixing declarative logic programming with novel communication abstractions yields a powerful framework for architecting next-generation distributed systems.

1. Introduction

Cloud computing is a vision for next-generation applications, which replaces the desktop-as-a-platform with the Internet-as-a-platform. Proponents of cloud computing envision that non-specialists should be able to build robust applications using the computational resources of the Internet. This is a departure from many of today's web applications, which may use a few reliable web services developed by industry experts (e.g. *Google Maps* or *Virtual Earth*). Economically, cloud computing provides new opportunities for large companies to lease their computing power, further incentivizing the trend.

Unfortunately, application development for the Cloud is a perfect storm of: (1) distributed systems programming, (2) web development, and (3) uncertainty due to the underlying compute fabric. A number of new technologies are addressing these challenges. For example, Amazon's *Elastic Cloud Compute* (EC2) service [17] helps developers distribute their applications across robust servers. Compiler technologies, such as *Volta* [11], simplify web development by splitting monolithic applications into browser-side and server-side parts. *Web services* [3] and *workflow modeling* [1] provide a paradigm for architecting and deploying web-based distributed systems. Recent analysis techniques for systems with dynamic creation of processes [4, 6, 12] model the dynamic agent topologies found in cloud computing.

However, the scope and scale of cloud computing means no single technology will solve all of the engineering challenges. Instead, a constellation of existing and developing technologies must be integrated during implementation. Currently, this key integration step is ad-hoc and has little tool support, thereby threatening the adoption of the cloud computing paradigm.

This paper addresses the challenges of building a complete cloud solution by developing:

- A holistic framework, called BAM, supporting all the phases of development: from specification to integration to testing, validation and deployment. This is accomplished via a novel abstraction layer, based on *logic programming (LP), for end-to-end specification of cloud applications*. BAM's declarative style decouples the specification from the implementation and permits many possible implementation strategies.

- A new implementation strategy, which combines state-of-the-art technologies with *code generators to produce full implementations from* BAM *specifications*. Our implementation stack incorporates recent advances in programming languages, databases, and web-development. Meanwhile, the code generation step makes extensive use LP to produce efficient implementations. First experiments show that BAM is a powerful approach for architecting cloud applications.

This paper is divided into the following sections. Section 2 describes the BAM approach and introduces the running

example. Sections 3, 4, and 5 present the formal semantics and specification language used for BAM. Section 6 discusses our implementation approach. Finally, we conclude in Section 7.

2. Specifying Clouds with BAM

BAM specifications are comprised of several layers of interrelated descriptions, encouraging an incremental specification approach. These layers are: (1) *The data layer*: The set of key data structures and data invariants. (2) *The operation layer*: The set of operations for manipulating data. (3) *The connectivity layer*: The communication policy among distributed nodes in a cloud. Each layer is written in a structured LP language called FORMULA [8], which has been designed for high-level specifications.

We introduce the formal semantics of BAM through a simple example, which we call the *document management system* (hereafter referred to as *DocMan*). The purpose of DocMan is to store a user's documents onto a server. Each user can edit original documents on a local client with or without connection to the server. Whenever a client is connected to a server, there should be attempts to update the server's copies to match the documents on the client. Though the DocMan example is simple, its core functionality can be found in many places.

- *Conference/manuscript systems*: These systems motivate our DocMan example. *Manuscript Central* and *EasyChair* are a few exemplars.
- *Email clients/servers*: Users create email drafts and reorganize email folders in a local client. These changes are periodically synced with a server.
- *Disconnected scenarios with rich clients*: Users modify shopping carts, online profiles, or blog entries (in a browser) while disconnected from the internet; data is synchronized with the server whenever possible. This capability is particular useful for mobile devices.

3. Modeling the Data

3.1. General Concepts

The foundation of any BAM application is a set of data structures used for data persistence and communication. Because BAM applications start from a specification, not an implementation, data structures are defined using the simple and semantically precise mechanism of *free algebras* (also called a *term algebras*). A free algebra consists of a set of constant values (like, 1, $true$ or nil) and free constructors. A free constructor f is an n-ary function; when applied to a list of n objects it constructs a new object. For example, if f is a binary function, then $f(1,2)$ is the application of f to the constants $1, 2$. Free constructors create objects so that the following property holds: Two objects are equal iff they are the same constant (e.g. the integer $1 = 1$) or if they were constructed by exactly the same sequence of free constructors:

$$\Big[f(f(1,1),2) = f(f(1,1),2)\Big] \neq f(1,f(1,2)).$$

Note that our objects are much simpler than than those in implementation languages. Each object is immutable, stateless, and has a unique representation.

We now define the abstract notion of a state. Let Υ be a set of free constructors and Σ be a (possibly infinite) set of constants, then $\mathcal{T}_\Upsilon(\Sigma)$ is the set of all objects that can be generated from constants and constructors. (Note that Σ is always included in this set.) We write $\mathcal{T}$ when the context is clear. Given a characterization of the objects, a *state* X is just a finite set of objects. A *finite run* r is a sequence of states:

$$r = X_1, X_2, \ldots, X_k, \quad \text{where } \forall\ 1 \leq i \leq k,\ X_i \subset \mathcal{T}.$$

A transition to a new state occurs whenever a finite number of objects are removed from or added to the current state.

States of a system can be classified using predicates. For example, $IsDeadlock(X_i)$ might decide if X_i is a deadlocked state. We employ LP (logic programming) to define these predicates. In LP these predicates are called *queries*. A query is not an arbitrary predicate, but one specified using a restricted form of LP. Examples of queries appear below.

```
1. domain Documents {
   /// Data types for documents
3.   Document : (String,String).
4.   Field    : (String,Any).
5.   DocCopy  : (String,String).
6.   FldCopy  : (String,Any).

   /// Data types for sets of docs
9.   Snapshot : (Id).
10.  MemberOf : (Any,Any).
...
```

Figure 1. Data structures for documents.

3.2. Specifying Data Structures

The core data structures of the DocMan system are *"document"* and *"copy of document"*. Specifying these structures requires a number of related free constructors. For this purpose, FORMULA provides an encapsulation mechanism

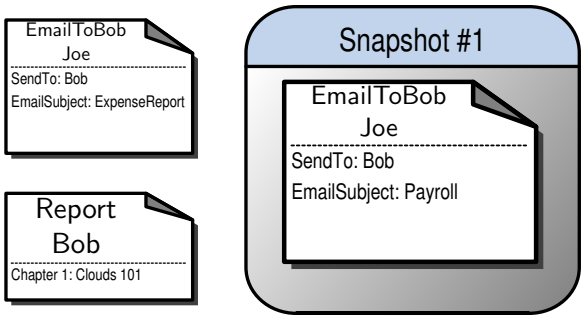

Figure 2. A sample state of the DocMan system.

that groups together constructors, clauses, and queries into a reusable package called a *domain*. Figure 1 shows a partial specification of these core structures. Line 1 declares a *domain* called Documents. Lines 3 to 6 declare constructors for documents and copies. FORMULA allows constructors to be partial functions with respect to an order-sorted type system. For example, Line 3 declares a binary constructor, called Document, defined only when both arguments are of type String. Every object has type Any; the Id type is a countably infinite alphabet of names (identifiers).

Each constructor has an intended use. The Document constructor creates a document object where the first argument should be the title of the document and the second should be the user who created the document, e.g. Document("Report", "Bob") represents a report created by Bob. The DocCopy constructor creates an object intended to be a copy of the similarly named document object. The Field constructor creates a name-value pair, where the name describes a field (of a document) having a particular value. The constructor FldCopy constructs fields that belong to document copies.

Other constructors are needed for relating documents, fields, and copies. The MemberOf function (Line 10) constructs membership objects serving this purpose. For instance the term

MemberOf(Field("Chapter 1", "Clouds 101"), Document("Report", "Bob")).

indicates that Bob's report has "Clouds 101" as its first chapter. Additionally, membership objects are used with *snapshot* objects (Line 9) to create sets of copies. Each server in the DocMan system has a snapshot object that holds many copies. For example

MemberOf(DocCopy("Report", "Bob"), Snapshot(#1))

indicates that the snapshot object with ID #1 contains a copy of Bob's report.

```
1.  model ExampleState : Documents {
      /// Some document objects
3.    Document("EmailToBob","Joe"),
4.    Document("Report","Bob"),
5.    Field("EmailSubject","ExpenseReport"),
6.    Field("Chapter 1","Clouds 101"),
      /// Some copy objects
8.    Snapshot(#1),
9.    DocCopy("EmailToBob","Joe"),
10.   FldCopy("EmailSubject","Payroll"),
      /// Some membership objects
12.   MemberOf(..Ln 5.. ,..Ln 3..),
13.   MemberOf(..Ln 6.. ,..Ln 4..),
14.   MemberOf(..Ln 9.. ,..Ln 8..),
15.   MemberOf(..Ln 10..,..Ln 9..),
      /// The remaining objects omitted
    }
```

Figure 3. State described with FORMULA.

Figure 2 illustrates a state of the DocMan system, ignoring the distribution of the data. In this state there are two original documents, called Report and EmailToBob. The snapshot contains only a copy of EmailToBob document and has an incorrect copy of EmailSubject field. Figure 3 shows this same state described as a finite set of objects using the FORMULA language. In FORMULA a finite set of objects is called a *model*. Line 1 declares that ExampleState is a model defined using the constructors of the Documents domain. Lines 12-15 construct membership objects using the shorthand ..Ln x.. which stand for *Use the same constructors found in line x*.

3.3. Clauses, Queries, and Invariants

The intended use of data structures must be explicit in the specification. This is accomplished by introducing a predicate $ViolatesUse(X_i)$, which examines the objects in state X_i, and returns $true$ if X_i violates the intended use of the data structures. If we view a state X_i as a database of objects, then this amounts to searching for some bad subset of objects. Logic programming provides a simple declarative mechanism for specifying these searches. Our logic programs are comprised of constants, free constructors and clauses, where a clause has the form:

$$h \leftarrow s_1, s_2, \ldots, s_n, \quad \neg t_1, \neg t_2, \ldots, \neg t_m. \quad \text{s.t. } n+m > 0 \tag{1}$$

The subexpressions h, s_i, and t_j are applications of free constructors, constant values, and, as we will see later, also of derived constructors; furthermore they may contain variables. The subexpression h is called the *head* of the clause; the remainder is called the *body*.

Evaluation of clauses can intuitively be understood as

follows: the body of a clause is a search criteria on objects in the state. A clause is executed by searching the current state for occurrences of the body. Each time a match is found, the head is activated. Activation causes a new object to be constructed from the head using the variable assignments that satisfied the body. This new object is *temporarily* added to the state, so that it can trigger other clauses. For instance, evaluating the following clause

$$hasACopy(t, u) \leftarrow Document(t, u), DocCopy(t, u). \tag{2}$$

against the state as defined in Figure 3 produces exactly one match with the assignments: t = "EmailToBob" and u = "Joe". This match results in the creation of the object:

hasACopy("EmailToBob","Joe")

recognizing that there exists a copy of Joe's email to Bob.

The body of a clause is partitioned into *positive* terms (the s_i's) and *negative* terms (the t_j's). Positive terms are *necessary patterns for a match*, and negative terms are *forbidden patterns*. If negative terms appear in the body, then the search criteria also requires the forbidden pattern to be absent from the state. For example,

$$notACopy(t, u) \leftarrow Document(t, u), \neg DocCopy(t, u). \tag{3}$$

creates a notACopy object for each document that does not have a copy. The observed behavior occurs because the positive part of the pattern is matched first. Variables fixed by the positive match restrict the forbidden pattern on those variables. Consider the following clause:

$$noCopies(u) \leftarrow Document(t, u), \neg DocCopy(y, u). \tag{4}$$

This clause creates a noCopies(u) object for each user u having zero copies assigned to u. In this case, the new behavior is caused by the variable y that is not restricted by the positive match $Document(t, u)$. We impose the usual constraint that each variable in the head h must appear in at least one positive term in the body.

Forbidden patterns have been studied in LP for many years under the moniker *LP with negation*. FORMULA uses a subset of LP called *non-recursive LP with stratified negation* [5]. This restriction is a syntactic property guaranteeing that no set of clauses will cyclically activate to generate an infinite number of objects; termination of the logic program is guaranteed. Another consequence is there exists an evaluation order for clauses such that: (1) The body of each clause needs to be examined for matches only once. (2) Negation is well-behaved - if a test for a forbidden pattern succeeds, then no later clauses will construct this forbidden pattern.

```
...
11. mayMember(x,y) :- x = Field(n,v),
12.         y = Document(t,u).

14. mayMember(x,y) :- x = FldCopy(n,v),
15.         y = DocCopy(t,u).

17. mayMember(x,y) :- x = DocCopy(t,u),
18.         y = Snapshot(id).

    /// Query for incorrect membership
21. badMember :? MemberOf(x,y),
22.         !mayMember(x,y).
...
```

Figure 4. Specifying the membership relation. (Con't from Figure 1.)

Henceforth we distinguish between objects that appear in the state of the system, and derived objects used to remember information about the state. Free constructors like Document and DocCopy denote objects in the state, derived constructors like hasACopy, notACopy, and noCopies, which are introduced via clauses, create only derived objects. Distinguishing between free and derived constructors originated in declarative databases [13]. In FORMULA free constructors start with a capital letter and their signatures have to be given explicitly. Derived constructors[1] start with a lowercase letter, and do not need to be explicitly declared.

A *query* is similar to a clause; it is an expression of the form:

$$Q \ :?\ s_1, s_2, \ldots, s_n, \quad \neg t_1, \neg t_2, \ldots, \neg t_m. \quad \text{s.t. } n+m > 0 \tag{5}$$

where Q is called the *query predicate* and the remainder is called the *body*. The predicate Q is evaluated over a finite set of objects X. $Q(X)$ evaluates to $true$ iff there exists at least one match for the body in X. Queries do not construct any new objects and cannot appear in the bodies of clauses.

We put these concepts together to specify the improper use of the constructors like MemberOf. Figure 4 shows the necessary FORMULA specifications, where ':-' denotes '$\leftarrow$', '!' denotes "$\neg$", and 'x = f(...)' allows 'x' to stand for 'f(...)'. The three clauses in Lines 11 - 18 thus construct mayMember(x,y) objects for each object x that could be a member of y, and lines 21 - 22 declare a badMember query that is $true$ whenever some membership object exists for which no corresponding mayMember is found. One might also wish to reject states with orphaned fields, but we omit these in the interest of space.

[1] Technically, derived constructors are still free constructors, in the algebraic sense. We use "derived constructor" as a shortened form of "derived free constructor".

The free constructors, clauses, and queries from Figures 1 and 4 can be derived directly from entity-relationship descriptions such as *UML class diagrams* [14], *metamodels* [2], or *database schema*. BAM provides automatic generation of these constructs. However, it is important to recognize that translation to structured LP provides a formal foundation useful for analysis [9] and code generation. Also, the logic program representation allows additional invariants to be incorporated into the specification that cannot be specified with a simple entity-relationship descriptions.

The most important part of the DocMan specification captures the expectation that documents are *synchronized* with copies. This part of the specification does not come from simple entity-relationship descriptions, and is particular to this application, as shown in Figure 5. Lines 26-27 define the missingDoc query, which returns $true$ if there exists a document without a copy. Notice the use of the variable s to search over all snapshots. Lines 37 - 42 define the missingFld and extraFld invariants, which detect if documents are no longer synchronized with the copies. This occurs whenever some copy is missing a field appearing in its original document, or has an extra field not appearing in its original.

Lines 44-45 expose a special query called unstable. This is a reserved query name for testing if a state violates the specification. This query is defined as a boolean combination of the other queries. Note that the ability to take boolean combinations of queries is also syntactic sugar. It would be possible to express the unstable query without this facility, by introducing additional clauses. Evaluating unstable on the state from Figure 3 returns $true$, because the snapshot is missing a copy of Bob's report and Joe's email has the wrong subject in the snapshot's copy.

```
...
23. hasDCopy(t,u,s) :- x = DocCopy(t,u),
24.     y = Snapshot(s),MemberOf(x,y).

26. missingDoc :? Document(t,u),
27.     !hasDCopy(t,u,s).

29. hasField(n,v,t,u) :-
30.     x = Field(n,v),
31.     y = Document(t,u),MemberOf(x,y).

33. hasFCopy(n,v,t,u) :-
34.     x = FldCopy(n,v),
35.     y = DocCopy(t,u),MemberOf(x,y).

37. missingFld :? hasField(t,u,n,v),
38.     !hasFCopy(t,u,n,v).

40. extraFld   :? hasFCopy(t,u,n,v),
41.     Document(t,u),
42.     !hasField(t,u,n,v).

44. unstable :? badMember or missingDoc
45.     or missingFld or extraFld.
46. }
```

Figure 5. Specifying invariants for DocMan. (Con't from Figure 4 .)

4. Evolving the State with Actions

Cloud applications require operations that manipulate data. These operations comprise the second layer of a BAM specification, and are defined with a special type of logic program called an *action*. Actions affect the state, in contrast to queries that only search the state creating transient objects along the way.

An action α has a signature of the form:

$$\text{action } \alpha \text{ [} p_1, p_2, \ldots, p_n \text{] : } D$$

where $[p_1, p_2, \ldots, p_n]$ is an optional list of *parameters*, and D is the name of a domain. An application of an action α to a state X is denoted:

$$\alpha \text{ [} o_1, o_2, \ldots, o_n \text{] } (X)$$

where $o_1, \ldots, o_n$ is a list of objects. The *body* of an action is similar to a domain; it is a set of free constructors and clauses, such that the logic program is non-recursive and stratified. Applying an action to a state causes a chain of events. First, a type-check verifies that the objects in X were built from the free constructors of domain D. Second, every occurrence of a parameter p_i in the logic program is replaced with the object o_i. Third, new objects are constructed by executing the logic program. Fourth, the objects generated by the program cause some objects to be added to X and some objects to be removed from X.

Figure 6 shows several actions. The first action (Lines 1-4) creates a new document with title t by user u. The action has one clause in its body; this clause checks that a document with the same title/user attributes does not already exist, and then creates an add(Document(t,u)) object. Since u and t are parameters, the clause is specialized with concrete values before the action executes. For example, NewDoc["ToDoList","Alice"](X) results in the following logic program:

$$add(Document(\text{"ToDoList"}, \text{"Alice"})) \leftarrow \neg Document(\text{"ToDoList"}, \text{"Alice"}) \quad (6)$$

containing no variables. The logic program executes normally, but accumulates special objects constructed with the unary add and del constructors. Let O denote all the ob-

```
1. action NewDoc[t,u] : Documents {
2.   add(Document(t,u)) :-
3.            !Document(t,u).
4. }

6. action DelDoc[t,u] : Documents {
7.   del(Document(t,u)) :-
8.            Document(t,u).
9. }

11. action CopyDocs : Documents {
12.   add(DocCopy(t,u)),
13.   add(MemberOf((DocCopy(t,u),x)) :-
14.       Document(t,u),x = Snapshot(s),
15.       !hasDCopy(t,u,s).
16. }
    /// Rest of the actions omitted
```

Figure 6. Three actions in the DocMan system.

jects constructed by the action, then at the end of execution the new state X' becomes:

$$X' = (X \cup \{s \mid add(s) \in O\}) - \{t \mid del(t) \in O\} \quad (7)$$

with the side condition that each s must be a valid (non-derived) object of the domain D. This *update* rule allows logic programs to effect the state in a predictable and modular fashion. Usage of the del constructor can be found in Lines 6-9. The CopyDocs action (Lines 11-16) is more complex. It searches for all pairs of documents/snapshots where the snapshot does not have a copy of the document. For every pair, a copy of the document is added to the snapshot. Note that this action uses a standard shorthand: Clauses with exactly the same body can be written as one clause with multiple head terms.

5. Characterizing the Cloud

Thus far our specification has intentionally avoided any issues of distribution over a cloud. Distribution is specified after the data and operation layers, encouraging a separation of concerns. At this final specification layer the user identifies:

1. A finite set of *agent types* A. The running system contains a heterogeneous mixture of nodes, each of which has exactly one agent type.

2. A *communication policy* $F \subseteq A \times A$ is symmetric relation on agent types. This policy determines if any two nodes u and w of agent type A_u and A_w can establish a point-to-point communication channel.

3. A *data access policy* $\rho : A \to \mathcal{P}(\mathcal{T})$ is a map from an agent type to a set of objects. This policy constrains the data that can be stored on a node u with agent type A_u.

In our abstraction, a cloud consists of a finite set of nodes connected by *ideal* (no information loss), *bidirectional*, and *point-to-point* communication channels. Formally, the *topology state* of a cloud is a finite undirected graph $G = \langle N, E \subseteq N \times N \rangle$. Each node is assigned an agent type according to $\tau : N \to A$. Nodes respect the communication policy; they can be connected only if the communication policy allows it:

$$\forall u, w \in N \quad (u, w) \in E \Rightarrow (\tau(u), \tau(w)) \in F. \quad (8)$$

In addition to the topology, each node has a *data state* according to $\delta : N \to \mathcal{P}(\mathcal{T})$, which assigns a finite set of objects to each node. Nodes respect the data access policy; they only contain objects allowed by this policy:

$$\forall u \in N \quad \delta(u) \subseteq \rho(\tau(u)). \quad (9)$$

A cloud application is a transition system that evolves through a sequence of topology/data states:

$$(G_0, \delta_0) \to (G_1, \delta_1) \to (G_2, \delta_2) \to \ldots \quad (10)$$

We shall refer to the pair (G_i, δ_i) as the i^{th} *cloud state* C_i. (Later in this section se discuss why it is sufficient to consider the cloud as a sequential system, despite the fact that cloud applications are distributed and concurrent.) The distribution of data is an essential property of distributed systems. The ability of the system to transition through different topologies is a fundamental characteristic of cloud computing. The challenge is to understand how an application behaves over this complex notion of state.

BAM addresses these challenges with a novel approach for extending the data and operation specifications beyond a simple state X_i to a cloud state C_i. First, the $unstable(X_i)$ predicate is generalized to $unstable(C_i)$ according to the following rule:

$$unstable(C_i) \stackrel{def}{=} \bigwedge_{M \in maxcliques(G_i)} unstable\left(\bigcup_{u \in M} \delta_i(u)\right) \quad (11)$$

where $maxcliques(G_i)$ denotes the maximal cliques in the undirected graph G_i. This rule declares that a cloud state violates the invariants of the data specification if there exists some maximal clique M, such that the aggregate data state on M would be found unstable by the basic logic program.

Figure 7 illustrates three topology states, with nine nodes per state; each node has type circle or box. These states differ in the connectivity between nodes resulting in different maximal cliques (shown in blue). State G_a is one extreme

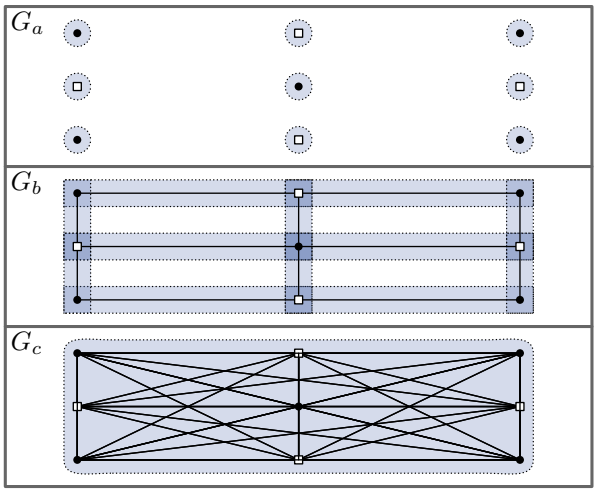

Figure 7. Extending the $unstable$ query over maximal cliques.

case where all the nodes are disconnected. The maximal clique semantics naturally assigns invariant evaluation locally to each node. State G_c is another extreme case where all nodes are connected. (Some edges overlap in the figure.) Every node is capable of directly transmitting information to every other node, so the maximal clique semantics evaluates invariants using the global data state of the system. All remaining topologies can be viewed as intermediate cases between these two extremes, as shown in state G_b. Note that a single node may participate in several evaluations of the $unstable$ query, if it occurs in multiple maximal cliques. For example, in G_b every node participates in more than one evaluation of the basic $unstable$ query.

5.1 Extending Actions over C_i

Next, we extend actions over cloud states. This task is more complicated, because the following circumstances must be taken into account: (1) The node that executed the action. (2) The information requirements of the action. (3) The data access policy restricting access to information. First, we characterize the *information requirements* of an action α in terms of the set of all non-derived objects that the action may observe, construct, or delete. Let $freecons(\alpha)$ denote the set of free constructors appearing in the body[2] of α . For example:

freecons(CopyDocs) = { Document, DocCopy, Snapshot,MemberOf }

[2]The body of an action may include extra clauses. For example, the CopyDocs action implicitly includes the clause from Documents having hasDCopy in the head. These extra clauses can be determined statically, and are used for calculating $freecons(\alpha)$.

Then, $\mathcal{T}_{freecons(\alpha)}(\Sigma)$ is the set of all objects that may be observed, constructed, or deleted by action α; we write $\mathcal{T}_\alpha$ for short. A node u *contributes* to the execution of α if it has access to some of the required information, i.e.

$$contrib(u,\alpha) \stackrel{def}{=} \quad \rho(\tau(u)) \cap \mathcal{T}_\alpha \neq \emptyset. \tag{12}$$

Though a single node may have access to some information, its access may not fulfill all the information requirements of α. When this happens, multiple communicating nodes need to pool their data access in order for the action to proceed. A set of nodes S *satisfies* the information requirements if:

$$satinf(S,\alpha) \stackrel{def}{=} \quad \mathcal{T}_\alpha \subseteq \left(\bigcup_{w \in S} \rho(\tau(w)) \right), \tag{13}$$

i.e., if S collectively has enough information rights to execute α. Let

$$\alpha\ [\ o_1, o_2, \ldots, o_n\ ]\ (u, C_i)$$

be the action α applied to cloud state C_i executed by node u. We define the pooling behavior with a set of nodes in the neighborhood of u that contribute information rights to the action α. This set is called the *horizon*:

$$horizon(u,\alpha) \stackrel{def}{=} \begin{cases} \{u\} \quad \text{if } satinf(\{u\},\alpha). \\ \text{otherwise:} \\ \{w | w \in N[u] \text{ and } contrib(w,\alpha)\}. \end{cases} \tag{14}$$

If u has enough rights for α, then the horizon is $\{u\}$. Otherwise, it is the set of nodes in the closed neighborhood of u (i.e. $N[u]$) that contribute to the access rights.

At this stage the useful nodes around u have been identified. However, the flexibility of the cloud permits the horizon nodes to be interconnected in arbitrary ways, as long as the communication policy is followed. Therefore, subsets of nodes must be located that can actually share information through direct communication. The sets are defined to be:

$$hmax(u,\alpha) \stackrel{def}{=} \left\{ M \;\middle|\; \begin{array}{c} M \in maxcliques(G_i[horizon(u,\alpha)]) \\ \text{and } satinf(M,\alpha) \end{array} \right\}. \tag{15}$$

These sets are the maximal cliques of the horizon's induced subgraph (i.e. $G_i[horizon(u,\alpha)]$) that also satisfy the information rights of α. The state-update equation becomes:

$$\delta'(w) = \begin{cases} \delta(w) \quad \text{if } \forall M \in hmax(u,\alpha),\ w \notin M. \\ \text{otherwise:} \\ \rho(\tau(w)) \cap \bigcup\limits_{M \in hmax(u,\alpha) \wedge w \in M} \alpha[o_1,\ldots,o_n] \left(\bigcup\limits_{v \in M} \delta(v) \right). \end{cases} \tag{16}$$

A copy of the action is spawned for each maximal clique in $hmax(u, \alpha)$, and the results are aggregated together. A node w can only remember the results allowed by the data access policy.

This novel communication semantics extends the logic program over an abstraction of the cloud. It incorporates data distribution, data access policies, communication policies, and topological dynamics. In our abstraction the point-to-point communication channels force nodes to explicitly announce their desires for non-local information. This naturally leads to a simple maximal clique semantics; a clique is a band of nodes whose members have agreed to work together. Finally, the horizon nodes and maximal cliques introduce symmetries into an otherwise non-symmetric setting; these are crucial for automated analysis [9].

Returning to the DocMan system, we can specify a cloud exhibiting the natural client-server architecture using a very simple specification. There are two agent types: $client$ and $server$, and communication is only allowed between clients and servers:

$$A = \{client, server\}. \quad F = \left\{ \begin{array}{c} (client, server), \\ (server, client) \end{array} \right\}. \tag{17}$$

The data access policy states that document and field objects are confined to clients, while snapshots, copies, and field-copies stay on servers.

$$\begin{array}{l} \rho(client) \mapsto \mathcal{T}_{\{Document, Field, MemberOf\}}(\Sigma), \\ \rho(server) \mapsto \mathcal{T}_{\{Snapshot, DocCopy, FldCopy, MemberOf\}}(\Sigma). \end{array} \tag{18}$$

Figure 8 shows the flexibility of our extension mechanism on a sample DocMan cloud. The boxes represent servers and the circles represent clients. The highlighted regions are the $hmax$ cliques resulting from various actions. The regions labeled α_1 result from the central server executing a single CopyDocs action. This synchronizing action causes pairwise communications between the server and the clients connected to it, resulting in the central server synchronizing with each client. No two clients exchange information with each other. Contrarily, α_2 is the same action, but executed by a client. In this case, only two pairwise actions are spawned as the client pushes data up to the servers. Finally, α_3 results from a client creating a new document. This action is localized to the client, since it has enough access rights on its own.

The provided semantics requires that each potentially distributed action is executed as a single transaction; parallel actions with intersecting information horizons are thus serialized. Actions with disjoint information horizons, however, can be executed in parallel. But, note that the state resulting from arbitrary serializations of a set of independent actions is always the same. This allowed us to view a cloud computation as a sequential transition system.

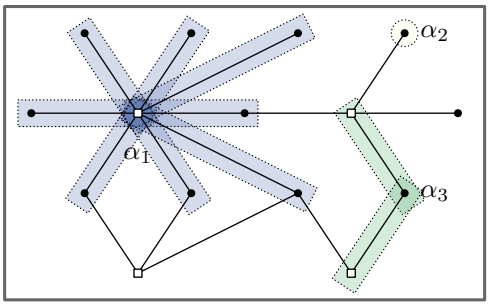

Figure 8. Extending actions over clouds.

6. Synthesizing BAM Applications

We have shown that BAM is a holistic specification language for describing the data, behaviors, and communications rules of a cloud application using structured logic programming and novel communication abstractions. Additionally, BAM has a declarative core and is thus decoupled from any particular implementation technologies, leaving ample opportunities for implementation approaches. On the other hand, BAM's rich abstractions mean that implementation is a non-trivial issue.

Moving from specification to implementation creates a number of real-world challenges:

- *Node discovery*: There is no centralized source that knows which nodes are in the cloud. Without this information, connections cannot be made between nodes and the system cannot perform useful work.
- *Maximal clique calculation*: Clique detection is a key aspect of the communication model. Nodes must collaborate to calculate cliques. It is particularly important to detect silently failing nodes, as these affect the maximal cliques.
- *Web interface*: Users expect a browser-based web interface, but this necessitates: XHTML, CSS, JavaScript, a stateful Web Server, and communication protocols.
- *Execution of queries and actions*: Queries and actions must be translated into an executable form. Also, non-local data must be brought onto whatever node executes some query or action.
- *Persistence layer*: Data needs to be persisted in a reliable manner.

In order to address these challenges, we view a BAM implementation as an integration of many technologies into one complete system. Of course, this is still leaves a non-trivial integration problem, but the integration problem can

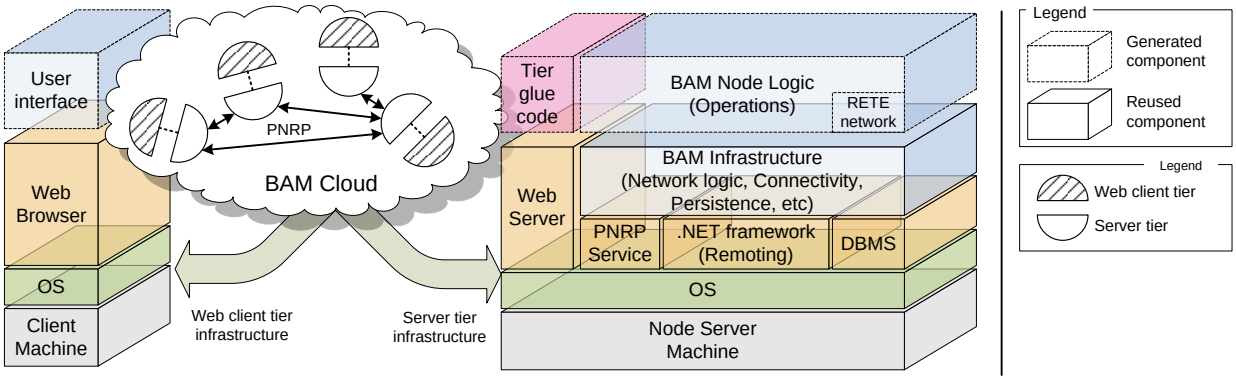

Figure 9. Architecture of synthesized cloud application.

be solved *once* for a chosen implementation stack and then reused. Code generators capture the integration strategy, and synthesize the necessary business logic on a per application basis. In the end, the user gets the best of both worlds: A state-of-the-art implementation stack based on known technologies and automatic generation of tedious and optimized business logic.

6.1. The Implementation Stack

Figure 9 shows the implementation stack and integration strategy. A BAM application consists of many communicating BAM nodes. Each node is itself split into two tiers: The *web client* tier and the *server* tier. The web client tier implements the browser-based web interface for a single node of the application. Each node also has its own server tier with a local stateful web server producing web pages for a browser. The server tier contains the majority of the implementation for each BAM node, and acts as a gateway from the browser to the larger cloud. (See the "cloud" in Figure 9.) The left-hand side of the figure summarizes the components of the web client tier, and the right side shows the server components. The dashed components are synthesized by code generation. We now discuss some of these components in more detail.

6.2. Node discovery

Node discovery is a central problem in *peer-to-peer networks*; we use an industrial-strength and widely-available discovery protocol for peer-to-peer networks called *PNRP* (Peer Name Resolution Protocol) [16]. PNRP is part of the Microsoft peer-to-peer infrastructure, and is present in all versions of Windows Vista and most versions of Windows XP. It allows nodes to be discovered with multicasting and requires only a few reachable servers for bootstrapping.

Each BAM node (repeatedly) uses the PNRP API to request the IP addresses of all other BAM nodes. This triggers an iterative process causing other machines running PNRP to pass this request throughout the network. Eventually, the request will arrive at nodes that have registered themselves as "BAM nodes" to the PNRP service. When this happens, the discovered BAM node will return its IP address, port information, and agent type to the requesting node. After a requesting node receives this information it can directly connect to the discovered BAM nodes.

6.3. Data persistence

A commercial database is the most obvious choice for a serious persistence layer; we use Microsoft SQL server for this purpose. However, proper configuration of the database requires:

- *Data definition scripts*: SQL scripts containing commands for table creation.
- *Data Objects*: The OOP counterparts of database tables. A program manipulates these objects, and then their changes must be persisted in the database.
- *Data Access Objects*: Objects that execute operations on the database (queries or updates). These objects perform tasks on behalf of the data objects.

Managing these various artifacts is notoriously tedious. We use a new programming language technology called *LINQ* (Language-Integrated Query) [10] that allows queries

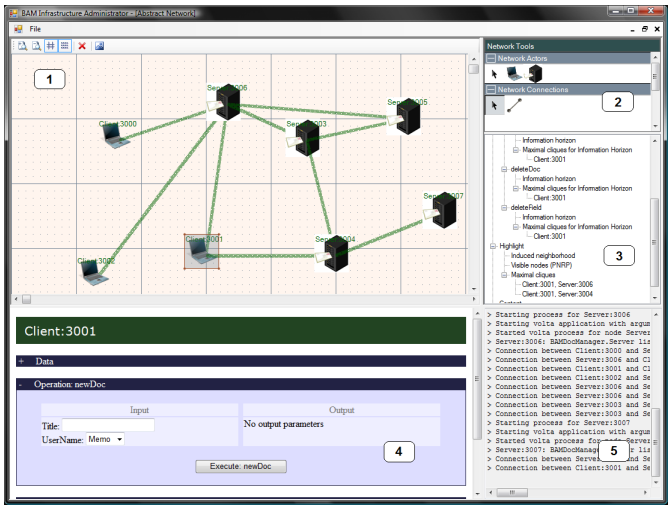

Figure 10. Synthesized monitor showing a running DocMan application.

to be embedded directly into a native programming language using data objects. LINQ generates the data access objects that interact with the database, and returns the results in form consumable as data objects. The code generator emits C# code with embedded LINQ queries to implement BAM actions, and also generates SQL scripts that initialize the SQL database.

6.4. Maximal Clique Construction

We use an incremental version of the algorithm presented in [15]. Every node u asks information about its neighbors' connections until u has a full picture of its induced neighborhood. Next, u locally calculates its maximal cliques from its local knowledge of its neighborhood.

More specifically, whenever a node u attempts to connect to a node w it also sends a list $N(u)$ of its neighbors to w. If w accepts u's connection request, then w adds a new edge (u, v) for each $v \in (N(u) \cap N(w))$ to its local knowledge of its induced neighborhood. Next, node w replies with a list of its neighbors $N(w)$. Node u updates its internal neighbor list, and then notifies its neighbors if new edges are created, and the procedure repeats.

The induced neighborhoods must also change whenever a node leaves the network. If a node exists gracefully, then it will notify its neighbors and they will remove this node from their lists. BAM nodes also send heartbeat messages to detect nodes that failed or became disconnected due to network connectivity. When a node detects a heartbeat failure, it will notify its neighbors of this failure and the induced neighborhood will be updated.

6.5. User interface

Each node should have an interface allowing the user to access its functionality; user expectations dictate that this should be a browser-based web interface. Two application tiers are needed for this: The *web client* tier, for interface presentation, and the *server tier*, for hosting web pages, executing node logic, and communicating with the larger cloud. We address these requirements using Microsoft Volta [11], a new compiler technology for automatic *tier splitting* of web applications.

Our code generators synthesize a single C# application where the methods have additional annotations. These annotations indicate which methods run in the browser and which methods run on the server. Next, the monolithic application is compiled into a .NET assembly with the annotations preserved at the binary level. The Volta compiler analyzes the byte code and converts the C# methods intended for the browser into JavaScript. GUI objects are also converted to XHTML objects backed by JavaScript and CSS. Next, a stateful web server is generated for the server tier. Finally, CGI-based communication between the client and server tiers is constructed. Volta significantly reduces the complexity of producing a realistic BAM application.

6.6. Other Components

Some other important components used in the implementation are .NET remoting, Rete networks, and templating-based code generators. The .NET remoting API is a mature technology for calling methods on remote objects. It uses an optimized binary communication protocol to pass data between remote objects, and uses proxy objects to make a remote object appear like a local one. BAM nodes

use PNRP to discover each other, but then switches to .NET remoting for all remaining communications.

Executing actions over a clique requires the nodes in the clique to elect a leader that will execute the action. Also, the other nodes in the clique must send relevant data to the leader before it can evaluate this action. In order to support this, we analyze the logic program and generate a dataflow network, called a *Rete network* [7], that incrementally calculates the result of an action as new data arrives. Thus, the body of actions are evaluated outside of the persistence layer, using synthesized code optimized for this task. The persistence layer is affected only after the results of the action are known.

Finally, the code generators are a combination of special purpose generators, e.g. the Rete synthesizer, and C# text templates. Text templating allows the majority of the generator to be written directly in the language that it emits. Additional control flow fragments are written in C#. This simplifies the code generators and improves maintainability.

6.7. DocMan Running on the Cloud

Figure 10 shows a bird's-eye view of the running DocMan system through a *cloud monitor*. This cloud monitor is automatically generated as part of the implementation, and displays the state of the cloud. Nodes can also be created/destroyed and connected/disconnected through the interface. Panel 1 reflects the current connectivity of the cloud. Different node types are assigned different icons: The black icons are servers and the gray icons are clients. Note, that this figure shows an extended version of the DocMan application that allows servers to directly communicate and exchange copies. Panel 2, to the right of 1, contains a list of generated network tools for modifying the state of the cloud. (The cloud can also evolve on its own.) Panel 3 shows the maximal cliques and information horizons calculated by the selected node, and Panel 5 displays messages passed between nodes. Finally, Panel 4 is an embedded web-browser for navigating to the web interface of an arbitrary node. The monitor can navigate to any of these generated web-pages.

7. Conclusion

In this paper we reported on a logic programming approach for specifying and synthesizing cloud applications. Our approach advocates a layered specification process, allowing engineers to reason about how their data invariants and actions will be extended over a cloud state. We also demonstrated one implementation approach employing a state-of-the-art technology stack with code generators. Finally, along the way we built a document managements system with a simple BAM specification, and then produced a full implementation from this specification. Our first experiments show that BAM is a powerful approach for end-to-end development of cloud applications.

References

[1] L. Aldred, W. M. P. van der Aalst, M. Dumas, and A. H. M. ter Hofstede. Communication abstractions for distributed business processes. In *CAiSE 2007*, pages 409–423, 2007.

[2] C. Atkinson and T. Kühne. Model-driven development: A metamodeling foundation. *IEEE Software*, 20(5):36–41, 2003.

[3] R. Breu, M. Breu, M. Hafner, and A. Nowak. Web service engineering - advancing a new software engineering discipline. In *ICWE05*, pages 8–18, 2005.

[4] L. Cardelli and A. D. Gordon. Mobile ambients. In *FoSSaCS 1998*, pages 140–155, 1998.

[5] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.

[6] E. A. Emerson and V. Kahlon. Parameterized model checking of ring-based message passing systems. In *CSL 2004*, pages 325–339, 2004.

[7] A. Gupta, C. Forgy, and A. Newell. High-speed implementations of rule-based systems. *ACM Trans. Comput. Syst.*, 7(2):119–146, 1989.

[8] E. Jackson, W. Schulte, and J. Sztipanovits. The power of rich syntax for model-based development. Technical Report MSR-TR-2008-86, Microsoft Research, June 2008.

[9] E. K. Jackson and W. Schulte. Compositional modeling for data-centric business applications. In *Software Composition*, pages 190–205, 2008.

[10] D. Kulkarni, L. Bolognese, M. Warren, A. Hejlsberg, and K. George. LINQ to SQL: .NET Language-Integrated Query for Relational Data. MSDN .NET Framework Developer Center (http://msdn.microsoft.com/en-us/library/bb425822.aspx), March 2007.

[11] D. Manolescu, B. Beckman, and B. Livshits. Volta: Developing distributed applications by recompiling. *IEEE Software*, 25(5):53–59, 2008.

[12] R. Marelly, D. Harel, and H. Kugler. Multiple instances and symbolic variables in executable sequence charts. In *OOPSLA 2002*, 2002.

[13] J. Minker. Logic and databases: A 20 year retrospective. In *Logic in Databases*, pages 3–57, 1996.

[14] Object Management Group. Unified Modeling Language: Superstructure version 2.1.1, http://www.omg.org/docs/formal/07-02-06.pdf, 2007.

[15] F. Protti, F. M. G. França, and J. L. Szwarcfiter. On computing all maximal cliques distributedly. In *IRREGULAR '97*, pages 37–48. Springer-Verlag, 1997.

[16] M. TechNet. Peer name resolution protocol. Microsoft TechNet (http://technet.microsoft.com/en-us/library/bb726971(TechNet.10).aspx), September 2006.

[17] J. Varia. Cloud architectures. Technical report, http://aws.typepad.com/aws/2008/07/white-paper-on.html, July 2008.