PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Morphe: A Constraint-Based Object-Oriented Language Supporting Situated Knowledge

Shigeru Watari, Yasuaki Honda, and Mario Tokoro*
e-mail: {watari, honda, mario}@csl.sony.co.jp
Sony Computer Science Laboratory Inc.
3-14-13 Higashi-Gotanda, Shinagawa-ku, Tokyo 141, Japan

Abstract

This article introduces Morphe, a programming language aimed to support construction of open systems. In open systems, the programmer cannot completely anticipate the future use of his programs as components of new environments. When independently developed systems are integrated into an open system, we eventually have inconsistent representations of the same object. This is because knowledge about the world is partial and relative to a perspective. We show how Morphe treats relative (and eventually inconsistent) knowledge by incorporating the notions of *situations* and *perspectives*.

1 Introduction

In modeling complex systems, one is often required to work with multiple representations of some aspects of reality. The notion of *situation* has been studied in computer science [Barwise 83][Barwise 89][Cooper 90] as an important concept in capturing the relative representation of knowledge about the world. The importance of such a notion stems from the epistemological assumption that any representation of the world is *partial* and *relative* to some perspective—that of the observer. In the cognitive process, the observer *abstracts* from reality only those aspects that he finds relevant; irrelevant portions are discarded. Sometimes this limited, abstracted representation is sufficient to allow one to perform certain tasks. In such cases we do not need to think about relative perspectives, and we can work as though our knowledge were an absolute and unique mapping of the real world. However, there are plenty of examples that show this is not true. In order to understand what is happening in the target world, we are forced to assume that the representation we are working with is relative, and furthermore, that we must eventually change perspectives in order to capture the real properties of the system we are representing. This is often the case when we have ambiguous representations and we are not able to resolve this ambiguity until we have some further information at hand.

Typically ambiguity arises when we try to combine information from different sources. For example, in dialogue understanding the knowledge of the one must be combined with the knowledge of the other to capture the exact meaning of an utterance [Numaoka 90]. Whenever there is some inconsistent information, the speakers must exchange further information in order to resolve the inconsistency. Other examples can be seen in multi-agent systems [Bond 88][Osawa 91]—where we have different agents with different knowledge bases that must be partially shared—and versioning systems as used in software development tools and engineering databases [Katz 90]—where we have different versions of the same object. A ground for extensive use of the notion of *situation* is in *open systems* [Hewitt 84], because in open systems the designer of a program cannot know *a priori* the nature of the environments in which their pieces of knowledge (called objects henceforth) will be used in the future. Along with its continuous evolution, an open system must be capable of integrating pieces of knowledge from different sources, and eventually these new pieces will conflict with existing ones.

In this paper we formalize the notion of *situation* as embedded in **Morphe**, a knowledge base and programming system which supports construction of open systems. *Situation* in Morphe is associated with a general notion of environment of interpretation. It represents a consistent set of properties (described by formulas) in a multi-version knowledge base. Rather than being a mere name for a part of the absolute real world, a *situation* has its own representation in Morphe, namely a routed,

*Also with Keio University, 3-14-1 Hiyoshi, Kohoku-ku, Yokohama, 223 JAPAN e-mail: mario@keio.ac.jp

directed, acyclic, and colored graph.

The notion of *situation* provides for two novel concepts: *compositional adaptation* and *situated polymorphic objects*. With *compositional adaptation* component objects are grouped within composite objects so that a component object is made to adapt to the requirements of the environment represented by the composite object. *Situated polymorphic objects* are objects that have multiple representations which depend on the situation they are used in. Situation is used to disambiguate the ambivalent interpretation of situated polymorphic objects.

The remainder of this paper is organized as follows: Section 2 gives an overview of Morphe's features through some examples. Morphe's formal syntax and semantics are sketched in Section 3 and Section 4, respectively. In this work we concentrate on the data modeling aspect of Morphe. Some important features (such as set-valued attributes, distinction between local and sharable attributes, user-defined constraints, and dynamic generation of new situations at update transactions) were not treated in the presentation for the sake of brevity and clarity. In Section 4 we give emphasis in showing how the domain of colored dags fits well to representing different perspectives to a shared object. In the last section we conclude this work.

2 Overview of Morphe

Morphe is a programming language which integrates object-oriented programming, constraint-based logic programming, and situated programming. It features:

- Querying capability for knowledge bases,
- Incremental construction of systems with inheritance and adaptive reuse of existent software,
- Multiple representations,
- Treatment of inconsistent knowledge through the notion of situation.

The basic aim of Morphe is to provide a system that supports easy construction of open information systems. There are two areas of support that are essential:

1. Easy integration of new pieces of knowledge, and
2. Treatment of shared inconsistent knowledge.

The Morphe system is a multi-version knowledge base with multi-versioned objects. We use the term *multi-version knowledge base* following the notion of *multi-version databases* as introduced by Cellary and Jomier in [Cellary 90]. Our approach differs from Cellary-Jomier's in that in Morphe even in a single knowledge base version we can have different object versions. The programmer can chose a particular version of the knowledge base through *situation descriptors*—formulas that index terms—which can be used within programs or in queries. In the development phase of a system, Morphe keeps track of transaction updates and creates consistent versions of the knowledge base.[1]

2.1 Example: Mario Joins Sony CSL

We will represent Sony CSL, a computer science laboratory, where Mario works as a director. We know that a representation of Mario already exists in the system and we want to share that representation. The existing representation is of Mario as a professor at an university.

```
1. person : [
       name : string;
       age : integer;
       sex : {male, female};
       age ≥ 0];
2. laboratory : [
       name : string;
       director : person;
       researcher :: person];
3. mario : person * [
       name : "Mario";
       age : 44];
4. scsl : laboratory * [
       name : "Sony CSL";
       director : person * [
              machine : "NEWS"]];
5. scsl.director = mario.
```

The first two expressions define the types for `person` and `laboratory`, and expressions 3 and 4 define `mario` and `scsl` as "instances" of `person` and `laboratory`, respectively. Expression 5 makes `mario` join `scsl` as its `director`.

Objects in Morphe are typed. For example, the expressions `name : string` and `age : integer` specify that the `name` of a `person` has type `string` and the `age` of a person has type `integer`. `String` and `integer` are primitive types provided in Morphe. The colon in those expressions represents a built-in predicate that specifies the type of the term on its left-hand side. Another built-in predicate is the one represented by the equal sign, as in `director = mario`, which specifies that `director`

[1] The operational aspects of manipulating situations are not emphasized in this work. Instead we will emphasize the declarative (or modeling) aspects of objects and situations.

and `mario` should have the same type. Expressions comprising these built-in predicates are called *formulas* or *constraints*.[2]

We can also construct complex types from primitive ones through *object descriptors*. An object descriptor is a set of formulas enclosed in brackets ("[]"). In the example, the expression `person` : [...] introduces a new type named `person` defined by the object descriptor on the right hand side of the colon.

As in unification grammar formalisms [Shieber 86] and some logic based programming languages [Kifer 89] [Yokota 92], Morphe does not make a distinction between classes and instances. Strictly speaking, every expression in Morphe is a type expression, and the execution of a Morphe program consists of finding the appropriate types for the variables, or in other words, solving the set of type constraints. Morphe provides domain specific constraint solvers and allows users to define predicates for new domains, as the predicate $\geq$ in the expression `age` ≥ 0. In this article we concentrate on showing how Morphe treats the notions of *situations* and *polymorphic objects*, leaving the discussion of other forms of constraints for another paper. Expressions using the colon predicate resemble attribute-value pairs of feature structure grammars and hence we sometimes refer, though improperly, to terms on the left-hand side of the colon operator as *attributes* and those on the right-hand side as *values*.

Besides object descriptors, there is another type of constructor: *braces* ("{}"). While object descriptors construct types intensionally, from formulas, *braces* construct types extensionally, from *terms*. For example, the expression `sex` : {`male`, `female`} specifies that the attribute `sex` of a `person` has type `male` or `female`. Stated in another way, the same expression defines a new type `person.sex` as a set of two constant types {`male`, `female`}.

A type can be made more and more specific as we add more restrictive constraints (formulas) into the associated object descriptor, and it becomes an "instance" when all the attributes are assigned constant types. In the code above, `scsl` is an instance of `laboratory` because the formulas in the object descriptor of the former are more restrictive than those in the object descriptor of the latter. Because all terms are types, even `scsl`, which is an "instance", can be made more specific by adding more formulas into its object descriptor. The way to do so is by *composing* object descriptors through `''*''`, the *composition* operator. The code which defines `mario` composes the type `person` with the object descriptor [`name` : `mario` ; `age` : 44]. The resulting object descriptor contains all the formulas of both operand types. The constraint solver then evaluates the most specific set of formulas in the resulting object descriptor, yielding [`name` : "`Mario`"; `age` : 44] as the type of `mario`. Determining the most specific sets of formulas is the same as determining the greatest lower bound of a set of terms. The associated procedure for determining the greatest lower bound is called *unification*, following the terminology of feature-structure grammar formalisms [Shieber 86].

2.2 Compositional Adaptation

With composition we can refine a type by giving more specific "values" for the attributes—as in `mario` above—or we can add new properties to an existing type. The type `laboratory.director` in the example is defined as a `person` plus an additional attribute: `machine`. Morphe allows for creating new types in a very particular way. The type `director` is defined in a specific context: `scsl`. This is an essential aspect of what we call *compositional adaptation*[Honda 92].

With compositional adaptation we make an object "adapt" to a new environment by transforming the object so that it obeys the type constraints specified in the environment. This process takes place when the predicate `''='''` is evaluated. When the expression `director` = `mario` is evaluated, it either succeeds or fails. If it succeeds, the object denoted by `scsl.director` is *unified* with the object denoted by `mario`, and the result of the unification can be accessed from both `scsl.director` and `mario`.[3] The object enters a new environment "acquiring" new properties and constraints. In the example, `mario` acquires the additional attribute `machine` as specified in the environment `scsl`, and `scsl.director` acquires all the original properties of `mario`.

2.3 Situated Polymorphic Objects

In programming languages, the term polymorphism has been traditionally associated with the capability of giving different things the same name. Morphe's notion of polymorphism follows in the same vein. In Morphe the

[2]The term "constraint" used here follows the terminology of constraint logic programming framework as formalized by Jaffar and Lassez in [Jaffar 87].

[3]The full version of Morphe allows programmers to specify which components of the type are private (i.e., local) and which are public (i.e., sharable). The public part of two objects must be compatible for the unification to succeed, while the private part is not affected in the unification.

same object can have different versions, eventually incompatible with each other. Incompatible versions of an object are called *morphes*, and objects that have multiple *morphes* are called *polymorphic objects*.

By incompatibility of morphes we mean incompatibility of their types.[4] Different morphes of the same (polymorphic) object may fundamentally mean two things: 1) different states due to updates, or 2) different representations due to different perspectives. Each *morphe* of a polymorphic object is situated. The evaluation of a polymorphic object is the evaluation of a *morphe*, the selection of which is subordinated to the selection of a *situation* where the object participates.

Each *morphe* is a consistent set of constraints that describe the behavior of the object in a given *situation*. For instance, a person may exhibit different and eventually contradictory behavior depending on the situation in which he acts. Inconsistent sets of constraints yield different values to be assigned to the same attribute. For example, suppose that the definition of `mario`, instead of that given in expression 3, had been: *mario* : *person* * `[name : ''Mario''; birthyear : 1947; sex : male; machine : ''Mac''];` After `mario` joins `scsl`, the attribute `machine` of `mario` is assigned the value `''News''` when he plays his role as `scsl.director` and a different value—`''Mac''`— in other situations.

2.4 Specifying a Situation

Morphe's notion of *situation* is tied to the notion of environment of interpretation. In the domain of interpretation, a *situation* is a graph representing the program being interpreted. *Situations* are used to disambiguate inconsistencies in the knowledge base. When an object participates in different environments (eventually created by independent programs) and is subject to independent transformations, it is often the case that the object must behave differently in each of them. Once the programmer wants a different view (or representation) for the object, the system creates a new version of the object in such a way that the situation is kept consistent.

When evaluating an expression within a situation, the system keeps track of the path through which the object containing that expression is being accessed. Access to an object from different *perspectives* is realized as different paths to the object. A *path* is a sequence of labels that allows one to navigate through the entire system, along the arcs in the graph. For example, if we want to refer to Mario when he plays his role of a director at SCSL we use the path `scsl.director`. *Paths* can be combined with formulas which filters the morphes of an object referred from the same path. For example, if we had several versions of Mario distinguished according to his age, we could access the representation of Mario at Sony CSL when he was at the age of 40 by using the expression: `scsl.director@[age = 40]`. We can also change the *perspective* by switching the path in the navigation. For example, we can switch the view from `mario` to `scsl.director` with the path `mario ↑ scsl.director`, which gives us the representation of `mario` from `scsl.director`'s perspective.

3 Syntax

The alphabet of Morphe consists of: 1) A: a set of *atoms*, 2) L: a set of *labels*, 3) X: an infinite set of *variables*, 4) the distinguished *predicate symbols*: ":" (*colon*) and "=" (*equal*), 5) the *composition* operator "*", 6) the logical connective ";", 7) the path constructors: ".", "↑", and "@"; 8) the auxiliary symbols "( )", "[ ]", "{ }", ",", and ".".

Atoms denote primitive indivisible objects. Example atoms are: `integer`, `string`, 3, and `''Mary''`. *Labels* are the names of the objects. The distinguished label `Home` denotes the topmost object in a particular situation.[5] In the semantic domain, the label names an arc which allows access to the objects down the (directed) graph.

3.1 Terms ($\mathcal{T}$)

Objects are denoted by *terms*. Terms are defined by:

$$\tau ::= x \mid a \mid p \mid [f] \mid \tau * \tau$$

where x are variables, a are atoms, p are paths, f are formulas, and $\tau * \tau$ are compositions.

The terms of the form $[f]$ are called *object descriptors*. Object descriptors construct complex objects through formulas, which are defined by:

$$f ::= p : \tau \mid \tau = \tau \mid f; f$$

A *colon* predicate is a typing constraint. An expression $e : t$, where e is a path and t is a term, specifies that the

[4] Informally, incompatible types means that the values of a type cannot be the values of the other. We give a formal definition of type incompatibility in the next section.

[5] Typically, the object denoted by `Home` represents the user's "home object", which is the user's entry-point into the Morphe system.

type of the object denoted by e has *at least* the properties defined by t. For example, the formula `mario : person` specifies that the `mario` has at least the properties specified by `person`.

The *equal* predicate specifies object sharing. Given $e_1 : t_1$ and $e_2 : t_2$, where e_1 and e_2 are paths, the expression $e_1 = e_2$ states that e_1 and e_2 denote the same object, and hence they have equal types. The shared object is "viewed" from different perspectives: any change to the object performed from a perspective must be reflected into other perspectives.

Because the atomic predicates *colon* (":") and *equal* ("=") impose a structure on the objects in the domain of interpretation (i.e., graphs), they are called *structural predicates*, in contrast to other domain predicates and user defined predicates. In this article we discuss only the structural predicates and hence we call them simply predicates.

A *path* names an object through a sequence of labels. Paths are defined by:

$$p ::= l \mid l.p \mid p \uparrow p \mid p@[f]$$

where l are labels. When an object is polymorphic due to different access paths, we select a morphe by the associated path. For example, in the subsystem:

$$\mathtt{a} : [\mathtt{b} : [\mathtt{c} : \mathtt{x}]; \mathtt{d} : [\mathtt{c} : \mathtt{y}]; \mathtt{a.b} = \mathtt{a.d}]$$

the polymorphic value of c can be disambiguated through the appropriate path: `a.b.c : x`, and `a.d.c : y`.

A path of the form $p_1 \uparrow p_2$ is a *path switch*. It allows one to view the same object from a different perspective. For example, the value of `a.b` $\uparrow$ `d.c` is `y`, instead of `x`.

A path of the form $p@[f]$ is called a *conditional path*. The formula enclosed in brackets on the right hand side of the @ sign is called a *situation descriptor*, because it specifies a family of situations which entail f. A conditional path has a meaning only in the situations where the formula enclosed in the brackets is entailed. For notational convenience we write $\mathtt{l} : \{\mathtt{t}_1@[\mathtt{f}_1], \mathtt{t}_2@[\mathtt{f}_2]\}$ instead of $\mathtt{l}@[\mathtt{f}_1] : \mathtt{t}_1; \mathtt{l}@[\mathtt{f}_2] : \mathtt{t}_2$. Conditional paths are used to select version morphes of polymorphic objects. For example, given

$$\mathtt{a} : [\mathtt{b} : \{\mathtt{x}, \mathtt{y}\}; \mathtt{c} : \{\mathtt{w}@[\mathtt{b} : \mathtt{x}], \mathtt{v}@[\mathtt{b} : \mathtt{y}]\}]$$

where `a`, `b`, and `c` are labels and `x`, `y`, `w`, and `v` are atoms, there are two possible values of `a.c`, which depend on the possible values of `b`. The formulas $\mathtt{b} : \mathtt{x}; \mathtt{c} : \mathtt{w}$) and $\mathtt{b} : \mathtt{y}; \mathtt{c} : \mathtt{v}$ determine two distinct situations of `a`. The value of `a.c` can be disambiguated by providing an appropriate conditional path: `a.b.c@[b : x] : w`, and `a.b.c@[b : y] : v`.

Composition is a binary operation $\mathcal{T} \times \mathcal{T} \rightarrow \mathcal{T}$ which composes two terms to produce a new term. Given two terms t_1 and t_2, their composition $t_1 * t_2$ is the union of the formulas contained in both terms. For example, [name : "John"; age : "integer"] * [age : 23] $\equiv$ [name : "John"; age : integer; age : 23].

3.2 Ordering on Terms

We have seen that terms denote objects in the intended domain, and formulas associate terms in order to represent complex structures in that domain. The *colon* operator specifies the structure of the object denoted by a given path. We can now amplify its use as a binary predicate over two terms to construct a partial ordering in the set of terms. We start with atoms. We assume that the atoms in A are partially ordered according to a binary relation represented by "$\leq_A$". For example: "*Mary*" $\leq_A$ *string*, and $3 \leq_A$ *integer*.

If $x \leq_A y$ and $y \leq_A x$ we say that x and y are *congruent*, and write $x \cong_A y$. The *greatest lower bound* of a set of elements $B \subset A$, denoted by $\downarrow B$ is defined as usual: $\downarrow B = inf \in A$ such that $\forall x \in B.\ inf \leq_A x$.

For notational convenience, we will denote the greatest lower bound of two atoms x and y by $x \downarrow y$. The greatest lower bound does not always exist. The elements c of A such that $x : c$ implies $x \cong_A c$ are called the *constants* of A.

We extend the partial ordering to the set of terms with the binary relation ":", defined by the rules below. In these rules, Γ is a set of formulas which defines a *situation*.

$$\Gamma \vdash x : y \quad (\text{ if } x, y \in A \text{ and } x \leq_A y)$$

$$\Gamma \vdash t : []$$

$$\Gamma, (e : t) \vdash e : t$$

$$\frac{\Gamma \vdash e@[\phi] : t}{\Gamma, \phi \vdash e : t}$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma, (e_1 = e_2) \vdash e_1 : t_3} \quad (t_3 = t_1 \sqcup t_2)$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma, (e_1 = e_2) \vdash e_2 : t_3} \quad (t_3 = t_1 \sqcup t_2)$$

$$\frac{\Gamma \vdash t_1 : t'_1 \ldots \Gamma \vdash t_n : t'_n}{\Gamma \vdash [l_1 : t_1; \ldots; l_n : t_n; \ldots; l_m : t_m] : [l_1 : t'_1; \ldots; l_n : t'_n]}$$

$$\Gamma \vdash t : t$$

$$\frac{\Gamma \vdash t_1 : t_2 \; \Gamma \vdash t_2 : t_3}{\Gamma \vdash t_1 : t_3}$$

The congruence relation on the set of terms is defined by: $x \cong y$ iff $x : y$ and $y : x$. The operation $\downarrow$ that gives the greatest lower bound of a set of atoms is also extended to terms. The rules below describe $\sqcup$, the greatest lower bound of two terms, defined so that $t_1 \sqcup t_2 : t_1$ and $t_1 \sqcup t_2 : t_2$.

$[] \sqcup t \cong t$

$x \sqcup y \cong x \downarrow y$

$[l : t] \sqcup [l : t'] \cong [l : (t \sqcup t')]$

$[l_1 : t_1] \sqcup [l_2 : t_2] \cong [l_1 : t_1; l_2 : t_2]$

$[l_1 : t_1; ...; l_n : t_n; l'_1 : t'_1; ...; l'_m : t'_m] \sqcup [l_1 : t''_1; ...; l_n : t''_n; l''_1 : t''_1; ...; l''_k : t''_k] \cong [l_1 : t_1 \sqcup t''_1; ...; l_n : t_n \sqcup t''_n; l'_1 : t'_1; ...; l'_m : t'_m; l_1 : t''_1; ...; l_n : t''_n; l''_1 : t''_1; ...; l''_k : t''_k]$

$t_1 \sqcup t_2 \cong t_2 \sqcup t_1$

$t_1 \sqcup (t_2 \sqcup t_3) \cong (t_1 \sqcup t_2) \sqcup t_3$

$t \sqcup t \cong t$

Two terms t_1 and t_2 are *incompatible* iff $t_1 \sqcup t_2$ does not exist.

4 Semantics

The formal semantics of Morphe is based on the algebraic approach to graph grammars as described in [Ehrig 86] and [Ehrig 90]. The domain of interpretation of Morphe is a set of colored, rooted, directed, and acyclic graphs. Following [ParisiPresicce 86][6], we impose a structure in the coloring alphabet in order to represent unification in that domain.

4.1 Definition: Colored Graphs

Let X be an infinite set of variables, A the set of atoms, L the set of labels (as introduced in Section 3), and O a set of identifiers. Let $C = (C_N, C_A)$ be a pair of alphabets where $C_N = O \cup A \cup X$ and $C_A = L$. The partial-order in A, $\leq_A$, is extended on C_N (and denoted $\leq_N$) such that $x \leq_N y$ iff $x \leq_A y$ or $y \in X$. A *C-colored graph* (or C-dag, for short) is a graph g over C defined as a tuple

$$< N_g, A_g, color^N_g, color^A_g, src_g, tgt_g, root_g >$$

where: N_g is the set of nodes; A_g is the set of arcs; $color^N_g : N_g \rightarrow C_N$ associates a color to each node; $color^A_g : A_g \rightarrow C_A$ associates a color to each arc; $src_g : A_g \rightarrow N_g$ associates with each arc a unique source node; $tgt_g : A_g \rightarrow N_g$ associates with each arc a unique target node; $root_g$ is a distinguished node called the *root* of the graph. It satisfies: $tgt^{-1}(root_g) = \emptyset$.

In what follows we refer to C-dags as graphs. A graph g is a *subgraph* of g' (written $g \subseteq_{\mathcal{G}} g'$) iff $N_g \subseteq N_{g'}$, $A_g \subseteq A_{g'}$, and the functions $color^N_g$, $color^A_g$, src_g, and tgt_g are the restrictions of the corresponding mappings of g'.

4.2 Definition: Graph Morphism

A *graph morphism* $f : g \rightarrow g'$ is a pair of functions $f_N : N_g \rightarrow N_{g'}$ and $f_A : A_g \rightarrow A_{g'}$ such that:

1. f_N and f_A preserve the incidence relations: $src(f_A(a)) = f_N(src(a))$ and $tgt(f_A(a)) = f_N(tgt(a))$,
2. f_A preserve the arc colors: $\forall a \in A_g.\ color^A_{g'}(f_A(a)) = color^A_g(a)$, and
3. $\forall x \in N_g.\ color^N_{g'}(f_N(x)) \sqsubseteq color^N_g(x)$.

A graph morphism indicates the occurrence of a graph within another graph. A graph morphism $f = (f_N, f_A)$ is called *injective* if both f_N and f_A are injective mappings, and it is called *surjective* if both f_N and f_A are surjective. If $f : g \rightarrow g'$ is injective and surjective it is called an *isomorphism*, and there is also an inverse isomorphism $f^{-1} : g' \rightarrow g$. In this case we say that g and g' are *congruent* and write $g \cong_{\mathcal{G}} g'$.

4.3 Subsumption

Subsumption is an ordering on graphs which corresponds to the relative specifity of their structures. A graph g *subsumes* h ($h \sqsubseteq_{\mathcal{G}} g$) iff there exists a graph-morphism $f : g \rightarrow h$ such that $f(root_g) = root_h$.

The semantic counterpart of the greatest upper bound of a set of terms (ref. Section 3.2) is the join of two graphs, which is their "most general unifier". The *join* of graphs g_1 and g_2 (notated $g_1 \sqcup_G g_2$) is a graph h such that $h \sqsubseteq_{\mathcal{G}} g_1$ and $h \sqsubseteq_{\mathcal{G}} g_2$.

4.4 Semantic Structure

The semantic structure of Morphe is a tuple

$$\mathcal{A} =< \mathcal{G}^*, \sqsubseteq_{\mathcal{G}}, \sqcup_{\mathcal{G}}, \top >$$

where:

1. $\mathcal{G}^*$, the domain of interpretation, is the set of all variable-free (i.e., ground) C-dags.

[6] F. Parisi-Presicce, H. Ehrig, and U. Montanari allowed variables in graphs (and productions) so that they could represent composition of graphs using relative unification. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, and M. Löwe [Corradini 90] further extended that work to represent general logic programs with hypergraphs and graph productions.

2. The relation $\sqsubseteq_{\mathcal{G}}$ and the operation $\sqcup_{\mathcal{G}}$ are as defined above.
3. *Top* ($\top$) is the distinguished element of $\mathcal{G}^*$ defined by: $\forall g \in \mathcal{G}^*.\ g \sqsubseteq_{\mathcal{G}} \top$.

4.5 Interpretation

A consistent set of formulas is represented with a C-dag with variables. The C-dag representation of a set of formulas is called a *situation*. A Morphe program is mapped by the interpreter into a set of situations which are ordered according to the subsumption relation. The evaluation of a query is a mapping from the C-dag representing the query to the set of *situations* in the hierarchy. If no situation is specified, the interpreter evaluates in a default situation. While parsing its input, the interpreter keeps track of this situation in order to resolve eventual ambiguities.

Let $\mathcal{I}_a : A \rightarrow C_N$ be a function that maps each atom in A to a node color in C_N, and $\mathcal{I}_\lambda : L \rightarrow C_A$ another function that maps each label to an arc color in C_A.

Variable Assignment

A variable assignment in a situation s is a mapping $\mu : X \rightarrow \mathcal{G}^*$ which maps variables to ground C-dags. We extend the variable assignment to other terms with the following clauses:

- If a is an atom, $\mu(s,a) = g$ *s.t.* $N_g = \{x\}, A_g = \emptyset$, and $color^N(x) = \mathcal{I}_a(a)$.
- If l is a label, $\mu(s,l) = g \sqsubseteq_{\mathcal{G}} s$ *s.t.* $\exists a \in A_s.\ color^A(a) = \mathcal{I}_\lambda(l)$ and $src(a) = root_s$ and $tgt(a) = root_g$.
- If l is a label, and e is a path, $\mu(s, l.e) = \mu(\mu(s,l), e)$.
- If e is a path and ϕ is a formula, $\mu(s, e@[\phi]) = \mu(s,e)$ if $s \models \phi$.
- $\mu(s, [\phi]) = g \sqsubseteq_{\mathcal{G}} s$ *s.t.* $g \models \phi$.

Formulas

The "truthness" of a formula is relative to a specific situation. We say that a situation s models a formula ϕ under a variable assignment μ (written $s \models_\mu \phi$) iff there is a subgraph of s with the properties specified by the formula.

$s \models_\mu e : t$ iff $\mu(s,e) \sqsubseteq_{\mathcal{G}} \mu(s,t)$.

$s \models_\mu e_1 = e_2$ iff $\mu(s,e_1) \cong_{\mathcal{G}^*} \mu(s,e_2)$.

$s \models_\mu \phi; \psi$ iff $s \models_\mu \phi$ and $s \models_\mu \psi$

5 Conclusion

This paper has shown how the notions of *situation* and *polymorphic objects* in Morphe can handle situated knowledge in open systems. We claim that the Morphe features shown here are suited to support incremental development of a complex system. When a set of constraints is added to a situation, the new formulas may conflict with the old ones. Morphe helps the developer to find the locus of inconsistency, and in the cases where the programmer wants a new version of the system, Morphe splits the inconsistent situation into new subsituations whenever it is possible. Some meta-rules based on domain-dependent heuristics may help the system to decide on which actions to take in the presence of conflict.

Syntactically, a situation was defined as a set of formulas which define a hierarchy of versions of the knowledge base. *Situation descriptors* can be used in programs in order to specify *a priori* the family of situations in which the program is expected to work. Once the system is provided with a way to determine the right situation, the associated *morphe* can be selected and then passed to the constraint solver in order to proceed with the evaluation of the program or the query.

Most existing typed programming languages impose a distinction between types and values syntactically, and types are usually associated with the variables in order to check whether the value assigned to a variable is compatible with the associated type. Morphe does not impose such a distinction at the syntactic level, though it bears both the notions of "types" and "values". An equal treatment of types and values was achieved in Morphe by imposing a partial order on the set of terms. This partial ordering was identified as the subsumption relation over directed acyclic graphs in the domain of interpretation.

In this work we have shown only those features that we find most interesting to capture the intuitive notion of relative knowledge, perspective, and situations. Problems concerning changes of situations in the presence of transaction updates, locality of information and sharing (i.e., unification), database querying facilities, and the operational semantics were not treated here. We hope however that the contents of this article have given the readers an insight on the problems and solutions concerning relative representations of objects in open systems.

Acknowledgments

Sony Computer Science Laboratory has been a privileged environment for discussing the problems and requirements of open distributed systems. Discussions with the

other members of this laboratory have provided the underlying motivations for developing Morphe. In particular, we wish to thank Ei-Ichi Osawa, for his collaboration at the initial phase of Morphe, and Akikazu Takeuchi and Chisato Numaoka for their helpful comments on the formalisms presented in this work. Watari thanks the members of Next-Generation Database Working Group promoted by ICOT. Discussions in the group promoted a better understanding of the requirements for advanced data base programming languages.

References

[Barwise 83] Jon Barwise and John Perry. *Situations and Attitudes.* The MIT Press, 1983.

[Barwise 89] Jon Barwise. *The Situation in Logic.* Center for the Study of Language and Information, 1989.

[Bond 88] Alan H. Bond and Les Gasser, editors. *Readings in Distributed Artificial Intelligence.* Morgan Kaufmann, 1988.

[Cellary 90] Wojciech Cellary and Geneviève Jomier. Consistency of Versions in Object-Oriented Databases. In Dennis McLeod, Ron Sacks-Davis, and Hans Schek, editors, *Proceedings of 16th International Conference on Very Large Databases*, August 1990.

[Cooper 90] Robin Cooper, Kuniaki Mukai, and John Perry, editors. *Situation Theory and its Applications – Volume I.* Center for the Study of Language and Information, 1990.

[Corradini 90] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, and Michael Löwe. Graph Grammars and Logic Programming. In *Proc. of the 4th International Workshop on Graph-Grammars and Their Application to Computer Science*, Springer-Verlag, March 1990.

[Ehrig 86] Hartmut Ehrig. Tutorial Introduction to the Algebraic Approach of Graph Grammars. In *Proc. of the 3rd International Workshop on Graph-Grammars and Their Application to Computer Science*, Springer-Verlag, December 1986.

[Ehrig 90] Hartmut Ehrig and Michael Löwe Martin Korff. Tutorial Introduction to the Algebraic Approach of Graph Grammars Based on Double and Single Pushouts. In *Proc. of the 4th International Workshop on Graph-Grammars and Their Application to Computer Science*, Springer-Verlag, March 1990.

[Hewitt 84] Carl Hewitt and Peter de Jong. Open Systems. In J. Mylopoulos and J. W. Schmidt M. L. Brodie, editors, *On Conceptual Modeling*, Springer-Verlag, 1984.

[Honda 92] Yasuaki Honda, Shigeru Watari, and Mario Tokoro. Compositional Adaptation: A New Method for Constructing Software for Open-ended Systems. *JSSST Computer Software*, Vol.9, No.2, March 1992.

[Jaffar 87] Joxan Jaffar and Jean-Louis Lassez. Constraint Logic Programming. In *Proceedings of the Fourteenth ACM Symposium of the Principles of Programming Languages (POPL'87)*, January 1987.

[Katz 90] Randy H. Katz. Toward a Unified Framework for Version Modeling in Engineering Databases. *ACM Computing Surveys*, Vol.22, No.4, December 1990.

[Kifer 89] Michael Kifer and Georg Lausen. F-Logic: A Higher-Order Language for Reasoning about Objects, Inheritance, and Scheme. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, ACM, 1989.

[Numaoka 90] Chisato Numaoka and Mario Tokoro. Conversation Among Situated Agents. In *Proceedings of the Tenth International Workshop on Distributed Artificial Intelligence*, October 1990.

[Osawa 91] Ei-Ichi Osawa and Mario Tokoro. *Collaborative Plan Construction for Multiagent Mutual Planning.* Technical Report SCSL-TR-91-008, Sony Computer Science Laboratory, August 1991.

[ParisiPresicce 86] Francesco Parisi-Presicce, Hartmut Ehrig, and Ugo Montanari. Graph Rewriting with Unification and Composition. In *Proc. of the 3rd International Workshop on Graph-Grammars and Their Application to Computer Science*, Springer-Verlag, December 1986.

[Shieber 86] Stuart M. Shieber. *An Introduction to Unification-Based Approaches to Grammar.* Center for the Study of Language and Information, 1986.

[Yokota 92] Kazumasa Yokota and Hideki Yasukawa. Towards an Integrated Knowledge Base Management System. In *Proceedings of the FGCS'92*, ICOT, June 1992.