KNAML: A Knowledge Representation Language for Distributed Reasoning

Gordon Streeter[1] and Andrew Potter[1]

[1] Sentar, 4900 University Square, Suite 8,
Huntsville, Alabama 35816 USA
`{gstreeter, apotter}@Sentar.com`
`http://www.sentar.com/`

Abstract. The Knowledge Agent Mediation Language (KNAML) is designed for use in multi-agent reasoning systems. Like conceptual graphs, KNAML represents knowledge using concepts, relations, and graphs. Concepts and relations are linked to form graphs, and graphs may be nested within other graphs. Additional constructs are used to support distributed reasoning and ontological concision. KNAML treats ontologies as knowledge domains that happen to be of the ontology domain. It uses an ontology of ontologies to define the concept and relation types available in an ontology. KNAML knowledge resources are modular to facilitate rapid development and efficient inter-agent processing. KNAML supports ontological specification of an extensible set of knowledge modalities, such as workflows, decision trees, and graphs that reflect the processing specializations of various knowledge agents and supports multi-modal knowledge authoring. Implemented in Java, KNAML supports subsumption, unification, and binding operations required by the host multi-agent system to carry out knowledge discovery and synthesis.

1 Introduction

Multi-agent systems that perform distributed reasoning pose distinct challenges for knowledge representation languages. Because the agents share a distributed knowledge corpus, the language must support explicit ontologies. To facilitate agent diversity, these ontologies should define, in addition to domain conceptualizations, various structural features of the language as well. Because contributions to multi-agent systems may come from a variety of organizations, the resources used to implement the language should use widely accepted technologies. Reliance on esoteric programming languages should be avoided. Even so, the knowledge representation language must readily support advanced reasoning capabilities, such as subsumption, unification, and binding. As with monolithic knowledge systems, the language should also support human readability and understandability. Finally, the knowledge representation language should support agent interoperability through the use of standard network transmission and data storage.

The Knowledge Agent Mediation Language (KNAML) is designed specifically for use in multi-agent reasoning systems. As with conceptual graphs [1], KNAML represents knowledge using concepts, relations, and graphs. Concepts and relations may be linked together to form graphs, and graphs may be nested within other graphs or concepts. Additional constructs, including concept frames and arc labels, are provided to support distributed reasoning and ontological concision. Frames are used to implement relational instantiation, which enables the system to provide discrete handling for factual assertions. Arc labels are syntactic helpers used to facilitate non-ambiguous mapping between ontological structures and knowledge content.

KNAML supports knowledge capture and agent specialization by implementing an extensible set of knowledge modalities, such as workflows, rules, decision trees, and graphs. This is accomplished by using an ontological specification for each modality. Each modality is accompanied by a corresponding editor that enables the user to create integrated knowledge projects, consisting of a set of multi-modal knowledge modules defined to address an anticipated range of problems using a multi-agent architecture. At the storage and transmission level, KNAML is represented as XML. KNAML is being used in a variety of applications now under development. Editors for workflows and graphs are now in use, and additional editors are in development.

This paper describes KNAML and its use in the KnoWeb® multi-agent architecture. The KnoWeb architecture uses mediated reasoning to integrate a variety of agent capabilities. These include knowledge bases, databases, UML workflows, and sensors. The Java implementation of KNAML supports subsumption, unification, and binding. The result is a simple but highly expressive language for representing knowledge for performing automated reasoning in a distributed environment.

2 Background: a Multi-Agent Architecture

KnoWeb is a multi-agent system architecture that uses mediated reasoning to perform dynamic decision-making [2]. KnoWeb employs a small group of core agents to implement its reasoning model, and engages a loosely coupled confederation of specialist agents to carry out goal-driven and event-driven tasks. The core agents consist of a Meta Agent, a service agent, and one or more domain advisors. The *Meta Agent* provides domain-neutral mediation and conflict resolution. In its role as mediator, the Meta Agent enlists other agents to satisfy goals presented by a requesting agent. It takes care of inter-agent coordination and planning needed to reach a goal. By concentrating reusable intelligence in this central resource, redundant complexity in specialist agents is reduced. The *service agent* maintains a registry of agent capabilities used to provide brokerage services. The service agent is functionally similar to matchmaking agents described in [3], however, in KnoWeb agents typically are both information providers and requesters, resulting in agent interaction that may be intensively cooperative. The *domain advisor* provides conflict resolution and planning strategies used by the Meta Agent. Agents communicate using the Knowledge Agent Mediation Language (KNAML) developed by Sentar.

KnoWeb mediated reasoning is implemented in Java. The Java implementation of KNAML supports subsumption, unification, and binding. Subsumption uses existential conjunctive logic to establish the truth value of one graph based on the known value of another. Graph and sub-graph unification supports discovery by indicating how one graph is subsumed by another. Binding is used for joining graphs to produce knowledge synthesis. The binding operation also supports backtracking. This is necessary to assure that graphs are recoverable in the event of binding failure.

The reasoning process used by the Meta Agent is straightforward. Throughout the process, the Meta Agent uses an agenda to keep track of what it is doing and why, and it maintains a context of asserted propositions. The process consists of several phases: initiation, alliance, marshalling, resolution, and response. The process begins when an agent initiates a request. The agent does so by sending the request to the Meta Agent. When the Meta Agent accepts a request, it first checks to see if the answer is already in the context or if the request is already in its agenda. If the answer is already in the context, the Meta Agent uses it to generate a response. Otherwise, it proceeds with problem solving.

In the alliance phase, the Meta Agent selects a relevant domain advisor for use in planning and conflict resolution. This alliance is sometimes necessary because the Meta Agent is domain neutral, and both planning and conflict resolution involve domain specific considerations. Typically, the domain agent is an expert system constructed specifically for a problem domain.

Following the alliance phase, the Meta Agent performs marshalling. Marshalling consists in identifying the agents to enlist in handling the request. Registered capability and ontological consistency are among the criteria. Also as part of marshalling, the Meta Agent negotiates with the domain advisor to determine whether any agents should be excluded from request processing. This affords the domain advisor an opportunity to eliminate extraneous branches from the agenda before it is executed by the Meta Agent. An agent's registered capabilities may include preconditions that must be resolved as part of request processing. These preconditions may require the Meta Agent to spawn additional requests, so the Meta Agent must be able to maintain recursive contexts.

Upon completion of marshalling, the Meta Agent dispatches the request to the enlisted agents. These agents, if they choose to handle the request, attempt to instantiate it, and return their results to the Meta Agent. The agents may initiate nested requests as needed, and the Meta Agent will invoke the reasoning process for each of these requests.

In the resolve phase, the Meta Agent discards duplicate responses and passes the remaining responses to the domain advisor for evaluation. The domain advisor may discard additional responses. The domain advisor may also initiate further nested requests which must be serviced prior to resolution of the original request. The remaining responses are asserted into the context and the solution is dispatched to the agent that initiated the request.

Note that elsewhere in the literature the term "Meta Agent" is used to refer to an agent that reasons about other agents [4] or as an agent that aggregates other agents [5]. Although this latter definition might have some functional applicability here, to the extent that multiple Meta Agents could operate among intersecting or complementary agent clusters, no other architectural support for this concept seems necessary. And for reasoning about other agents, no particular kind of agent is required. What would be required are appropriate knowledge resources and some stock of problems to consider.

3 Knowledge Agent Mediation Language

KNAML is based on conceptual graphs, as defined by Sowa [1]. In KNAML, knowledge is represented using concepts, relations, and graphs. Concepts and relations may be linked together to form graphs, and graphs may be nested within other graphs. Ontological support is built into the language. The result is a simple but expressive language for the representation of complex knowledge.

A concept may represent any entity. Concepts have a type and a referent. The type is the ontological category to which instances of the concept belong. The referent denotes a specific instance or set of instances of a concept. The following graph contains a single concept. The concept type is `Person` and the referent is `#Bob`:

```
[
      [Person:#Bob]
]
```

In the above example, the type of concept referent used is called an indexical. Indexicals are always preceded by the "#" sign, e.g. `#Bob`. Indexicals may be used to designate individual concepts. KNAML supports two other kinds of referents. These are string literals and descriptors. String literals are represented using character strings enclosed in quotes. Descriptors are graphs. These are used frequently in KNAML. In the following example, the referent of the concept type `Proposition` is a descriptor. As shown, descriptors can be nested:

```
[
      [Proposition:[
            (Believes)
                  +-believer-->[Person:#Jack]
                  +-belief-->[Proposition:[
                        (GoingTo)
                              +-traveler-->[Person:#Bob]
                              +-destination-->[City:#Boston]
                  ]]
      ]]
]
```

Frames are implemented using relations, with arcs for each slot. Frames are used for relational instantiation. This makes it possible for the system to distinguish one otherwise identical instance of a relation from another. If there are multiple assertions, for example, that "Bob is going to Boston," possibly received from differing agents or from the same agent at different times, relational instantiation permits the system to uniquely identify each trip. Further, frames provide a convenient way to specify properties for each instance, such as time and date or mode of transportation. So, if the ontology for concept type `Person` specifies associations with concepts of type `Address, City, State, and Zip`, the association can be defined like this:

```
[Person: #Bob [
      (PersonalDetails)
```

```
          +-name-->[Name: "Robert McNamara"]
          +-address-->[Address: "1000 Defense Pentagon"]
          +-city-->[City: "Washington"]
          +-state-->[State: "DC"]
          +-zip-->[Zip: "20301"]
]]
```

Relations are used to define relationships among concepts. Each relation has a type and a collection of arcs. The arcs are used to define the linkage between a relation and its concepts. Arcs are labeled. The labels are determined in the ontological definition of the relation. Each arc terminates in a concept. The types of each of these concepts are also specified ontologically.

Here, the relation `GoingTo` has two arcs, one labeled `traveler` and the other `destination`. The `traveler` arc terminates on a concept of the Type `Person`, with referent of `#Bob`. The destination arc terminates on a concept of type destination, with a referent of `#Boston`.

```
[Proposition:[
     (GoingTo)
          +-traveler-->[Person:#Bob]
          +-destination-->[City:#Boston]
]]
```

Suppose that several people are going to Boston. One way to express this would be to create several graphs, one for each traveler. Another would be to define the ontology to permit the use of a plural. A plural is a special concept in which all the elements are of a specified type:

```
[
   [Proposition: [
       (GoingTo)
            +-travelers-->[Person: { #Bob, #Carol, #Ted,
                                     #Alice }]
               +-place-->[City: #Boston]
   ]]
]
```

A graph is a container for concepts, relations, and other graphs. A graph is represented by a matching set of square braces. Here is a graph containing a relation and its ontologically specified set of concepts:

```
[

    (GoingTo)
        +-traveler-->[Person:#Tex]
        +-destination-->[City:#Madison]
]
```

4 Ontology

As previously suggested, ontology is central to KNAML. Use of an explicit ontology enforces consistency within a knowledge module, and more importantly for multi-agent systems, ontology makes it possible for agents to share knowledge. An ontology is a specification of the concepts comprising a domain and the interrelationships that may hold between these concepts. We treat an ontology as a knowledge module whose domain happens to be an ontology. To author such an ontology, an ontology-ontology is required.

An `Ontology` consists of an ontology name and a collection of concept and relation types. The name is a concept of type `TypeName`, and the concept and relation types are of `ConceptType` and `RelationType` respectively:

```
[Ontology:[
     (OntologyFrame)
          +-aName-->[TypeName:]
          +-someConceptTypes-->[ConceptType:{*}]
          +-someRelationTypes-->[RelationType:{*}]
]]
```

A concept type is a concept of type `ConceptType`. To define a concept type, the `ConceptType` concept type is used. The `ConceptType` has two parts, a type name and a type. The name is a concept of type `TypeName`, and the type is ontologically unspecified and could be anything. By convention, the type must be a relation type, an enumeration, a primitive, or nothing at all.

```
[ConceptType:[
     (ConceptTypeFrame)
          +-aName-->[TypeName:]
          +-aType-->[]
]]
```

A relation type is a concept of type `RelationType`. A relation type specification consists of a name and a collection of arc types. The name is a concept of type `TypeName`. The arcs are concepts of type `ArcType.`

```
[RelationType:[
     (RelationTypeFrame)
          +-aName-->[TypeName:]
          +-someArcTypes-->[ArcType:{*}]
]]
```

An arc type is a concept of type `ArcType`. The specification of an arc type consists of a label, an arc terminus, and a plural indicator. The label is a concept of type `TypeName`. The terminus is a concept of type `ConceptType.` The plural indicator is a concept of type `Boolean`.

```
[ArcType:[
     (ArcTypeFrame)
          +-aLabel-->[TypeName:]
          +-aTerminus-->[ConceptType:]
          +-aPluralIndicator-->[Boolean:]
]]
```

There are several primitive types used to provide the ontological foundation. These include `TypeName`, `Boolean`, and `Enumeration.` A concept of type `TypeName` is a string, a concept of type `Boolean` may be either true or false, and a concept of type `Enumeration` may be used to define a specific set of values that may be applied to a concept.

Using the ontology-ontology, it becomes possible to treat an ontology as a knowledge module. This means new ontologies can be authored using the same tools used to create other knowledge modules, and further, it is possible for a knowledge agent to reason about ontologies just as they do about other domains. We anticipate being able to apply this approach to ontology reusability problems.

5 Knowledge Modalities

In a multi-agent system, agent specializations may occur along lines of knowledge domains. For example, one agent might specialize in some area of product diagnostics, and another could specialize in customer service. The two agents combined would be useful in creating a product support application. However, there is another form of specialization, one which occurs along lines of knowledge modalities. That is to say, the agents specialize in their forms of knowledge representation. Some problems are best solved by using rules, others by using decision trees, and still others respond well to workflows. The possibilities are unlimited. A single form of knowledge representation, no matter how powerful, is insufficient. Using the wrong representation leads to poor design, difficult knowledge authoring, and poor system performance.

KNAML supports an extensible set of modalities, such as workflows, rules, decision trees, and graphs. This is accomplished by using KNAML ontological support to create ontologies for specialized modalities. Each modality is accompanied by a corresponding editor that enables the user to create integrated knowledge projects, consisting of a set of multi-modal knowledge modules defined to address a predefined range of problems using a multi-agent architecture.

For example, the workflow modality allows knowledge to be authored, expressed, and processed as workflows, using workflow symbology. The workflow agent implements UML activity diagrams, including actions, forks, merges, branches, joins, and transitions. Workflow activities and transition guards include goals, which are expressed as KNAML graphs. At runtime, these goals are evaluated—using the Meta Agent reasoning process where appropriate—and the results are used to determine the path taken by the workflow. That multi-agent systems would benefit from a well-defined agent interaction protocol is clear [6]. The workflow agent orchestrates the behavior of the multi-agent system, and it does so in an architecturally neutral manner. A workflow is a tactical plan for solving a problem. By specifying the steps required to solve the problem, the order in which they are to be taken, and the conditions under which they will be invoked, the workflow provides a coherent approach to agent cooperation. Because all activities performed by other agents (possibly including other workflow agents) are mediated through the Meta Agent, workflows can maintain goal-level visibility into the problem solving process. This simplifies the knowledge representations required by individual agents and reduces the need for extensive preconditions on agent capabilities. Supporting the workflow agent is a workflow editor used to create workflow modules.

Thus, in addition to support knowledge processing, the multi-modal approach makes knowledge representations more intuitive for non-logicians. We believe this is a significant benefit, especially in contrast to knowledge representations which rely exclusively on description logics, markup languages, or some combination of the two, as is the case with the Resource Description Framework, as described in [7] and elsewhere.

6 Reasoning in KNAML

The Java implementation of KNAML supports subsumption, unification, and binding. Subsumption uses existential conjunctive logic to establish the truth value of one graph based on the known value of another. Graph and sub-graph unification supports discovery by indicating how one graph is subsumed by another. Binding is used for joining one graph with another. The binding operation also supports backtracking.

We have chosen to implement KNAML using Java. This commitment is consistent with our general ground-rule that our development be portable, practical and accessible. Implementing unification in Java has required that subsumption, unification, and binding be addressed explicitly, whereas, in a logic programming language such as Prolog, these capabilities might have been left to fend for themselves. Here we discuss some of the issues associated with defining and implementing the KNAML reasoning capability.

6.1 Subsumption

Subsumption is used for graph comparison. For example, it may be used to compare a possible solution to a goal. If the goal can subsume the possible solution, then the possible solution is, in fact, a solution. This technique has been used in the Meta Agent to check a new goal against the context, to see if the solution is already known. Subsumption is a very specific test for similarity in two conceptual structures. A conceptual structure `p` is said to subsume the structure `q` if the following conditions hold:

1. If `p` and `q` are concepts, then `p` subsumes `q` if the type of `p` subsumes the type of `q` and the referent of `p` subsumes the referent of `q`.
    a. The type of `p` subsumes the type of `q` if the former is unspecified or if the two are identical.
    b. The referent of `p` subsumes the referent of `q` if any of the following are true:
        i. The referent of `p` is unspecified.
        ii. The referents of `p` and `q` are identical primitives.

        iii. The referents of `p` and `q` are subsumable plurals.[1]
        iv. The referents of `p` and `q` are subsumable graphs.

2. If `p` and `q` are relations, then `p` subsumes `q` if the type of `p` is identical to the type of `q` and the arcs of `p` subsume the arcs of `q`. The arcs of `p` subsume the arcs of `q` if all of the following are true:
    a. The number of arcs in `p` is equal to the number of arcs is `q`.
    b. There is a one-to-one match from the arc labels in `p` to the arc labels in `q`.
    c. The concept at the end of each arc with a given label in `p` subsumes the concept at the end of the arc with the same label in `q`.
3. If `p` and `q` are graphs, then `p` subsumes `q` if one of the following conditions hold:
    a. The content of `p` is unspecified.
    b. Both `p` and `q` are empty.
    c. `p` and `q` are non-empty, and for every structure in `p`, there is a corresponding structure in `q` which the structure in `p` subsumes.

6.2 Unification

Unification supports discovery by producing a new graph which shows how one graph is subsumed by another. In the Prolog programming language, both unification and subsumption lend themselves to backtracking. For example, the matching of graphs is not order dependent, so an attempt may begin in one order until it fails, then another order is attempted. However, there is no backtracking in Java—once an object is changed, it is changed. Therefore the Java implementation of unification does not change the arguments being unified, but produces a third, unified argument. If at any point during the unification of `p` and `q` to produce `r`, some substructures `p′` and `q′` are being unified in an attempt to produce `r′` and the attempt succeeds, `r′` can be added to `r`. But if it fails, any structures accumulated into `r′` can be discarded, and any other appropriate attempt can be made.

Interestingly, when `p` is unified with `q` to produce `r`, it would seem that `r` would be identical to `q`. We might draw this conclusion from a cursory reading of the rules for subsumption, which say that `p` is subsumes `q` if `p` is identical to `q`, or, in some specific instances, if `p` is unspecified. In the instances in which `p` and `q` are identical, `r` will have the same value as `p` and `q`, and so, obviously, it will have the same value as `q`. In the instances where `p` is unspecified, `q` may be specified or unspecified, but in either case, `r` will have the value of `q`. So, in all cases, `r` will have the same type and structure as `q`. What unification allows `r` to inherit from p is indexicals.

[1] The definition for subsumption of plurals is not given here, but it is essentially the same as subsumption of graphs.

Consider the case of an agent that can convert from degrees Fahrenheit to degrees centigrade. To make the agent capability declaration simple, let us say that the agent can convert both directions, and also can check a given pair of numbers to see if they are paired by the conversion process (that is, that the conversion of one would result in the other). The first of the following three propositional functions represents this capability, the second represents a goal for the conversion of 32 degrees Fahrenheit to centigrade, and the final propositional function represents the unification of the two:

```
[
    [PropositionalFunction:#p[
        (Convert)
            +-degreeF-->[#theFarValue]
            +-degreeC-->[#theCentValue]
    ]]
    [PropositionalFunction:#q[
        (Convert)
            +-degreeF-->["32"]
            +-degreeC-->[]
    ]]
    [PropositionalFunction:#r[
        (Convert)
            +-degreeF-->[#theFarValue"32"]
            +-degreeC-->[#theCentValue]
    ]]
]
```

In keeping with the earlier discussion, these are labeled "`#p`", "`#q`", and "`#r`", respectively. Notice that unification takes the values from `q`, the goal, which is supplied by the system, but retains the indexicals from `p`, the capability, which was supplied by the agent. This allows the agent to use the indexicals to quickly locate important concepts in the goal, even though unification may have a different form from the original capability. Granted, it would not be difficult to locate any concept in this example. Actual capabilities are generally not quite so simple.

6.2.1 Unspecified graphs. A unique requirement that has emerged in our use of KNAML is the need to be able to work with unspecified graphs. The KNAML code has for some time been aware of "null" graphs, graphs which have not yet had their element vector set. In various places, the code attempts to treat these graphs the same as empty graphs. The concept of unspecified graphs is borrowed from plurals, where there is both the empty plural "{}" and the unspecified plural "{*}". Sowa [1, 8] does not include this notion in his definition of graphs, and is silent on the whole idea of propositional functions. Further, if graphs can be made to be a true superset of functionality to plurals (and why should they not?) then plurals would be redundant, and could be replaced by sequences. Sequences are similar to plurals, but are ordered. These could be important, for example, when we wish to send a list of options to the user, and we wish those options to be presented to the user in the same order they were sent.

6.2.2 Unification by Sub-graph. Unification by sub-graph is important as a vehicle for discovery and a test of truth. What is true for a graph should also be true for a sub-graph. For example, consider a goal p formed from the question, "Who is going to Boston and Chicago?" The proposition q, presumably from context, shows that it is known that Bob is going to Baltimore, Boston, Chicago, and Washington:

```
[PropositionalFunction:#p [
      (Going)
           +-traveler-->[#theTraveler]
           +-destination-->[[
                ["Boston"]
                ["Chicago"]
           ]]
 ]]
 [Proposition:#q [
      (Going)
           +-traveler-->["Bob"]
           +-destination-->[[
                ["Baltimore"]
                ["Boston"]
                ["Chicago"]
                ["Washington"]
           ]]
 ]]
```

Obviously, if Bob is going to Baltimore, Boston, Chicago and Washington, then Bob is going to Boston and Chicago, and p should unify with q, as shown here in r:

```
 [Proposition:#r [
      (Going)
           +-traveler-->[#theTraveler"Bob"]
           +-destination-->[[
                ["Baltimore"]
                ["Boston"]
                ["Chicago"]
                ["Washington"]
           ]]
 ]]
```

6.3 Binding

Because unification does not affect p or q, it is very useful and relatively simple to implement. However, there are times when its usefulness is limited for this very reason. We have found we need an operation where p is "bound" to q, where the process of binding is like unification, except that the result, rather than being directed to a new structure r, is directed back into p. An example of this is found in the behavior of the workflow agent.

Consider a section of a workflow that monitors the temperature of some piece of equipment. As a failsafe, this section of the workflow takes two different temperature readings, one in Fahrenheit and one in centigrade, and compares the two to see if they agree. Presumably, if they do not agree, the workflow would generate some alarm, but we will concern ourselves only with the states in the workflow which take the two readings and compare them. Here are the goals associated with the states of interest:

```
[
      [PropositionalFunction:#goal1 [
            [Sensor:#theSensor[
                  (Property)
                        +-aValue-->[#degreesF]
                        +-aName-->["DegreesFahrenheit"]
                        +-aConcept-->[#theSensor]
            ]]
      ]]
      [PropositionalFunction:#goal2 [
            [Sensor:#theSensor[
                  (Property)
                        +-aValue-->[#degreesC]
                        +-aName-->["DegreesCentigrade"]
                        +-aConcept-->[#theSensor]
            ]]
      ]]
      [PropositionalFunction:#goal3 [
            (Convert)
                  +-degreesF-->[#degreesF]
                  +-degreesC-->[#degreesC]
      ]]
    ]
```

Note that the goals are joined. That is, the third goal shares concepts with the first two. When the third goal is executed, it needs the values returned from the execution of the first two goals. The workflow agent is an abstract agent, and knows nothing about any particular domain, so it certainly will not know that it needs to obtain these values for this particular case. The agent could be implemented to unify the goal with the result, then search future goals for indexicals found in the unified result, copying the value from the unified result to those goals, but this would be a messy process.

If, instead, the agent would, after execution of each goal, bind the goal with the result, future goals which are joined to the goal would automatically be changed. For example, if the Fahrenheit reading were 32, after the execution of the first goal and the binding of the goal with the result, the concept with the indexical “degreesF” would be bound to the value “32”. Similarly, if the reading for degrees centigrade were “0”, after the execution of the second goal and the binding of the goal with the result, the concept with indexical “degreesC” would be bound to “0”. This would set the third goal up to test the conversion of 32 degrees Fahrenheit to 0 degrees centigrade, which is exactly what we would want.

6.3.1 Binding and Subsumption. Since binding is destructive, it is useful to know in advance if it will succeed. Although it might seem that subsumption could be used as an indicator of success, there are problems with such an approach. Binding, like unification and subsumption, is not order dependent. The algorithm must attempt to find an order in which the graphs will bind. The subsumption and unification algorithms can simply try progressive orders until one succeeds, or until all have failed. Unlike subsumption and unification, binding changes the graph while in the process of trying progressive orders. This means that binding may not succeed, even when subsumable order has been found. Thus we have found it necessary to implement backtracking for binding.

7 Conclusion

In this paper we have described KNAML and its use in the KnoWeb multi-agent architecture. KNAML supports knowledge capture and agent specialization by implementing an extensible set of modalities, such as workflows, rules, decision trees, and graphs. This is accomplished by using an ontological specification for each modality. Each modality is accompanied by a corresponding editor that enables the user to create integrated knowledge projects, consisting of a set of multi-modal knowledge modules defined to address an anticipated range of problems using a multi-agent architecture. At the storage and transmission level, KNAML is represented as XML. KNAML is being used in a variety of KnoWeb applications now under development. Workflows and graph editors are now in use, and additional editors are in development.

The KnoWeb architecture integrates a variety of agent capabilities. These include knowledge bases, databases, UML workflows, and sensors. The Java implementation of KNAML supports subsumption, unification, and binding. Subsumption uses existential conjunctive logic to establish truth value of one graph based on the known value of another. Graph and sub-graph unification supports discovery by indicating how one graph is subsumed by another. Binding is used for joining graphs to produce knowledge synthesis. The binding operation also supports backtracking. This is necessary to assure that graphs are recoverable in the event of binding failure. The result is a simple but highly expressive language for representing knowledge for performing automated reasoning in a distributed environment.

References

[1] J. Sowa, Knowledge Representation: Logical, Philosophical, and Computational Foundations. Pacific Grove, CA: Brooks/Cole, 2000.

[2] G. Streeter, A. Potter, and T. Flores, A mediated architecture for multi-agent systems, presented at Seventeenth International Joint Conference on Artificial Intelligence: Workshop on E-Business and the Intelligent Web, Seattle, WA, 2001.

[3] K. Sycara, M. Klusch, S. Widoff, and J. Lu, Dynamic service matchmaking among agents in open information environments, ACM SIGMOD Record, vol. 28, pp. 47-53, 1999.

[4] J. Dix, V. S. Subrahmanian, and G. Pick, Meta Agent Programs, Journal of Logic Programming, vol. 46, pp. 1-60, 2001.
[5] J. Sutherland and W.-J. van den Heuvel, Enterprise application integration and complex adaptive systems, Communications of the ACM, vol. 45, pp. 59-64, 2002.
[6] G. W. Mineau, Representing and enforcing interaction protocols in multi-agent systems: An approach based on conceptual graphs, presented at IEEE/WIC International Conference on Intelligent Agent Technology, Halifax, Canada, 2003.
[7] S. Decker, Melnik, M., Van Harmelen, F., Fensel, D., Klein, M., Broekstra, J., Erdman, M. Horrocks, I., The Semantic Web: The Roles of XML and RDF, in IEEE Internet Computing, 2000, pp. 63-74.
[8] J. Sowa, Conceptual structures: Information processing in mind and machine. Reading, MA: Addison-Wesley, 1984.