FIRE: Infrastructure for Experience-based Systems with Common Sense

Kenneth D. Forbus[1], Tom Hinrichs[1], Johan de Kleer[2], Jeffrey Usher[1]

Qualitative Reasoning Group, Northwestern University[1]
2133 Sheridan Road, Evanston, IL, 60201, USA
{forbus, t-hinrichs, usher}@northwestern.edu;

PARC, Inc. [2]
3333 Coyote Hill Road, Palo Alto, CA, 94304, USA
dekleer@parc.com

Abstract

We believe that the flexibility and robustness of common sense reasoning comes from analogical reasoning, learning, and generalization operating over massive amounts of experience. Million-fact knowledge bases are a good starting point, but are likely to be orders of magnitude smaller, in terms of ground facts, than will be needed to achieve human-like common sense reasoning. This paper describes the FIRE reasoning engine which we have built to experiment with this approach. We discuss its knowledge base organization, including coarse-coding via *mentions* and a *persistent TMS* to achieve efficient retrieval while respecting the logical environment formed by contexts and their relationships in the KB. We describe its stratified reasoning organization, which supports both reflexive reasoning (Ask, Query) and deliberative reasoning (Solve, HTN planner). Analogical reasoning, learning, and generalization are supported as part of reflexive reasoning. To show the utility of these ideas, we describe how they are used in the Companion cognitive architecture, which has been used in a variety of reasoning and learning experiments.

Introduction

The hallmarks of common sense reasoning are that it is flexible, robust, and efficient. Deductive reasoning is flexible, but to date, is neither robust nor efficient. Small errors in knowledge can lead to an arbitrary number of incorrect conclusions, if proof by contradiction is not carefully controlled. Moreover, the NP-hard nature of deduction makes it scale badly. This in turn requires imposing tight resource bounds, making conclusions hit-or-miss, since partial results are not always enlightening. This suggests to us that common sense reasoning will not be achieved by a simple combination of deduction plus a large body of knowledge. There is a deeper problem as well. General, first-principles axioms excel at specifying what is logically possible in the domain being formalized. They do not capture as well what tends to actually happen in the domain. Given limited knowledge, the set of models implied by general axioms will always be much larger than the reality they are modeling. What is missing, we think, is experience. By experience, we mean knowledge of the kinds of entities there are, and what kinds of things have actually happened. If we can reason by analogy from experience, we should be able to flexibly reason about new situations, because analogy supports partial matches. If we can generalize based on accumulated experience, we can construct probabilistic models of the kinds of entities there tend to be and the kinds of things that tend to happen. These generalizations should provide robustness, since they are grounded in the statistical properties of the domain. Finally, if analogical matching, retrieval, and generalization can be made efficient, with tightly constrained deductive reasoning used to "fill in" small gaps, then reasoning can be done efficiently. This is the *experience-based* approach to common sense reasoning that we are exploring.

How much experience might be necessary for human-level common sense reasoning? An upper bound is extremely difficult to estimate, since it relies on a large number of assumptions about human processing capabilities. So let us estimate a lower bound. How much knowledge is in books? In a recent experiment (Lockwood & Forbus, 2009), an average of 8 assertions was generated for each simplified English sentence by a natural language understanding system, and an average of 388 assertions/diagram were generated by a sketch understanding system. For an eight page chapter, heavy in diagrams, this led to just over 10,000 assertions being generated. This suggests that for a 100 page book, we can expect on the order of 125,000 assertions. (If the book is text-only, this number might drop to around 32,000 assertions.) Let us take the average of 78,500 as an estimate. If a child in school reads 10 books/year for grades 1-12, encoding just this much knowledge is 9.4×10^6 assertions, or several times the current size of the ResearchCyc KB contents. If we use the diagram-heavy estimate, to start to take into account the kind of experience needed to ground what is needed to understand what is in the books, this number rises to 1.5×10^7, over an order of magnitude larger than ResearchCyc. Clearly

experience-based systems will place extreme demands on whatever infrastructure is used for reasoning.

The sheer scope of knowledge required indicates that hand-coding is out of the question. The need to garner experience from interaction, from human artifacts (e.g., books and the web) and people places another constraint on the reasoning infrastructure: Experience-based systems are best thought of as organisms rather than tools, since they will need to accumulate knowledge over extended periods of time, and continue to adapt as the distribution of what they encounter changes.

This paper describes our FIRE reasoning engine, which we created to support experiments in building experience-based systems. As described below, we have models of analogical matching, retrieval, and generalization that have both been used to model a wide variety of psychological results and have been used in a variety of performance systems. This paper does not focus on them, but on the knowledge base infrastructure and support for other types of reasoning that are needed for effective analogical reasoning and learning at scale. We start by describing the knowledge base (KB) organization, including the use of coarse-coded indexes for retrieval, a persistent TMS for logical environment support, and persistent case libraries to support analogical retrieval and generalization. Next we describe the reasoning services that FIRE provides, including how we stratify them for efficiency and trade off between reflexive and deliberative operations. How FIRE is used in the Companions cognitive architecture is then discussed, and we close with related and future work.

Knowledge Base Organization

We assume standard Cyc terminology in this paper, since the KB contents we use are derived from either ResearchCyc or OpenCyc, depending on the application, plus our own extensions. The essentials are: *Collections* model concepts, with `isa` indicating concept membership, e.g. `(isa Nero Mammal)`. Superordinate relationships between concepts are indicated by genls statements, e.g., `(genls Mammal Animal)`. Superordinate relationships between predicates are indicated by `genlPreds` statements, e.g., `(genlPreds touches near)`. Facts are stored in *microtheories*, which supply a form of context. Microtheories are related via inclusion using `genlMt` statements, e.g. `(genlMt HumanActivitiesMt AffectMt)`. A small number of predicates, such as `genlMt`, indicate facts that are believed in every microtheory, i.e., are *global*. All other facts are contextualized. Reasoning is always performed with respect to some microtheory. That microtheory and those it inherits from via `genlMt` statements form the *logical environment* for that reasoning.

The standard key operations for a KB include pattern-directed retrieval and structural inferences (e.g. answering questions about category inclusion, `genls` inferencing in Cyc parlance). Analogical processing requires two additional services. First, all of the facts which mention a particular individual need to be retrieved, as a core operation in dynamic case construction (Mostek et al. 2000). Second, it must support the persistent stores needed by analogical retrieval and generalization. Analogical retrieval requires storing *case libraries*, into which particular situations are added (Forbus et al. 1995). Analogical generalization requires storing *generalization contexts*, into which particular situations are added for incremental learning of probabilistic generalizations (Friedman & Forbus, 2009). We start by describing fact retrieval, then turn to analogical processing support.

Retrieving facts

FIRE's KB infrastructure is implemented using a persistent object database, AllegroCache[1]. Collections, predicates, entities, and microtheories are all implemented as CLOS objects, with structural facts involving them (e.g., `genls`, `genlPreds`, `genlMt`, and argument constraints) implemented via pointers involving numerical IDs assigned to each object. Other facts are stored as fact objects, with their own ID space. The difference in manner of storage is hidden behind a procedural interface: Retrieval operations dispatch on the predicate to either invoke the general-purpose retrieval system or specialized retrieval procedures relying on the pointer structure used to implement special facts. Many structural inferences, such as finding `genls` or instances, are implemented via procedures that use pointer operations for efficiency.

Coarse-coding for retrieval. The need to retrieve all facts which mention an entity as well as all facts matching a pattern requires some subtlety to achieve both efficiency and storage economy. McDermott (1975) argued that indexing schemes can be viewed as storing partial information about assertions, e.g., a list of all the assertions whose first element is `touches`, chopping up an assertion in all possible ways so that the constant parts of a query pattern can be used to find candidates. These candidates are then combined, for example, by intersection, and then subsequently filtered by unification to produce matches. Our experience with Cyc-sized knowledge bases indicates that this model requires some modification to operate at scale. First, intersection is rarely worthwhile, since it is typically faster to simply run unification on the smallest bucket. Second, exhaustive coding of all positions of an assertion can generate a massive number of buckets, almost none of which are useful. For example, using a 2006 snapshot of ResearchCyc, an exhaustive positional indexing scheme generated 701,692 buckets – the longest path through assertions was 104 elements long. The distribution of paths through assertions has a long tail: the vast majority of facts are described within a path length of seven. This suggests using a coarse-coded indexing scheme. We use only position within the top-level structure of an assertion for encoding, and treat anything beyond the sixth element as being equivalent.

[1] http://www.franz.com/products/allegrocache/

To implement buckets, we exploit AllegroCache's ability to do efficient range-based retrievals of sortable data structures, in this case, integers. Recall that all entities, predicates, collections, and microtheories are implemented as objects with a common ID space, and facts are implemented as objects with a second ID space. Suppose a fact with ID *F* mentions something with ID *E* anywhere in the *Nth* position. We encode this fact as a single integer by using the top bits for the entity ID, followed by 3 bits to encode the position, followed by the fact ID. This is illustrated in Figure 1.

S bits entity ID	3 bits loc	Fact ID

Layout of mention integer, 64 bits assumed

S	# entities	# Facts
32	4.3×10^9	5.4×10^8
28	2.7×10^8	8.6×10^9
24	1.7×10^7	1.4×10^{11}

Figure 1: Mentions indexing scheme for retrieval

Given a pattern to retrieve, two mentions integers are generated for each constant within it. Each integer starts with the integer ID for the KB object corresponding to that constant plus its coarse-coded position within the assertion, plus either all zeros for the fact ID or all ones. These integers constitute the upper and lower bounds for a ranged search, and the unique fact IDs for the mention integers within this range constitute the contents of one bucket. The smallest bucket is used to generate the candidate list for subsequent unification and filtering via logical environments (described next).

Mention integers are also designed to support rapid retrieval of all facts mentioning an entity, as needed for dynamic case construction. In that case, the location bits are also set to zeros or ones to provide the bounds for the ranged search, with the results subsequently filtered via logical environment.

Both of these operations are quite fast in practice – orders of magnitude faster than our prior scheme, which used a more standard table-oriented database. And, as the numbers in Figure 1 indicate, 64 bit integers should suffice for experimenting with knowledge bases several orders of magnitude larger than currently exist.

Persistent TMS for logical environment support. The logical environment used for a computation can contain hundreds or even thousands of microtheories. Thus it is crucial for efficiency to be able to quickly test whether or not a fact is part of the current logical environment. To do this, we adapt ideas from truth maintenance to create in the KB a *persistent TMS* (PTMS) which tracks changes in `genlMt` statements, so that testing membership of a fact with respect to a logical environment is quick.

The PTMS implementation is inspired by the ambiguity packing approach of the ATMS and HTMS (de Kleer, 1986; 1992). From an ATMS perspective, every microtheory is an ATMS assumption node. But there is only one ATMS environment per node. Every (`genlMT mt1 mt2`) introduces a justification `mt2`→`mt1`. As with TMS' the `genlMT` structure may contain cycles. Each microtheory has a label consisting of a single environment which contains itself, and all other microtheories it inherits from. Figure 2 describes a set of 6 `genlMT` statements and the resulting justification and label structure.

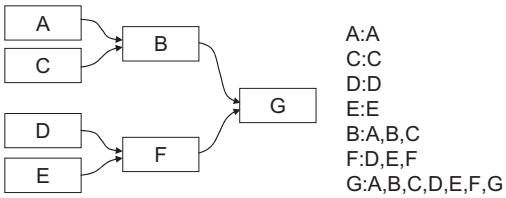

Figure 2: Justification and label structure resulting from `(genlM B A) (genlMT B C) (genlMT F D) (genlMT F E) (genlMT G B) (genlMT G F)`.

Labels are compactly represented by sparsely encoded bit vectors, which provide compact storage and rapid testing of whether labels overlap. Every fact in the KB has a label, indicating the set of microtheories it is believed in. Thus testing whether a fact is believed within a logical environment is carried out by intersecting the bit vectors corresponding to their labels. A compact structure representing new microtheories in the WM is transparently tied to the PTMS labels in the KB, so that all reasoning correctly and efficiently respects the constraints imposed by whatever logical environment is current.

Persistent storage for analogical retrieval and generalization. We view FIRE as an infrastructure for creating software organisms. That means they need to have persistent memories over time. We store cases as microtheories in the knowledge base. Case libraries are implemented by another type of KB object, which tracks what cases are considered to be part of the case library. As with other special facts, making or retracting assertions in the KB about case library membership causes these underlying pointers to be changed appropriately. Facts about case library membership are global, i.e., they do not depend on the current logical environment, since they are viewed as part of an organism's memory. The MAC/FAC analogical retrieval model (Forbus et al. 1995) uses *content vectors* in the first-stage processing, followed by analogical matching for the second stage. Content vectors are stored with cases and incrementally updated as facts are added to or removed from microtheories. (Content vectors are only computed for microtheories that are added to a case library, for efficiency.) Our previous scheme required reifying case libraries in FIRE's working memory, which was extremely inefficient and often caused heap blowout.

Our analogical generalization model, SAGE[2] (Friedman & Forbus, 2009), stores cases in *generalization contexts*.

[2] SAGE is the next step in evolution of our SEQL model.

For example, in modeling spatial language learning, each spatial preposition has its own generalization context, to which examples are added. SAGE compares incoming examples with a set of examples and generalizations that it maintains, assimilating the new example into a generalization if it is sufficiently similar, or starting a new generalization if it is sufficiently similar to one of the examples. Generalization contexts are implemented as objects that are a subclass of case libraries. This means that what to compare against in this assimilation process can be performed via MAC/FAC, instead of the previous exhaustive comparison algorithm.

A case library can also be specified dynamically as the union of a set of case libraries. This is useful for classification, e.g., MAC/FAC can be used over the set of generalization contexts for concepts. It can also be used to generate negative examples automatically (McLure et al 2010), by using all generalization contexts except for the target concept to retrieve close competitors.

Stratified Reasoning Services

Reasoning with a very large knowledge base requires both fast and efficient inference mechanisms and flexible, deliberative control strategies. These goals tend to conflict. We resolve this dilemma with two levels of reasoning: *reflexive inferences* that are highly constrained and run to completion, and *deliberative inferences* that can explicitly represent and reason about the control state and operate incrementally.

Reflexive Reasoning

The reflexive reasoning services in FIRE are designed to support "immediate" or "obvious" inferences. Operations at this level should not hang or loop forever, but should instead succeed or fail quickly. To ensure this, the reflexive level embodies several design decisions:

- It *stratifies* reflexive services into back-chaining inferences and the primitive, single-step inferences on which they are built.
- It is built on top of a logic-based TMS that provides, among other things, a fast *working memory* cache for intermediate results.
- It exploits the *microtheory context* mechanisms provided by the KB and the Cyc ontology to limit search.
- It *outsources* specially defined predicates as a principled way of escaping to Lisp and integrating external packages, such as analogy and spatial reasoning.
- It supports a number of *advice wrappers* to declaratively specify different kinds of resource limits on queries.

The reflexive level has two API calls that define the stratified services: **`ask`** and **`query`**. **`Ask`** is the most primitive inference, and in the simplest case just invokes the KB to retrieve statements and justifies them in the TMS (see below). It also supports microtheories, outsourced predicates, and advice wrappers. **`Query`** on the other hand, is the entry point for backchaining. It interprets rules in the form of Horn clauses and searches breadth-first for all solutions to the query. The use of Horn clauses is another way the reflexive level trades off flexibility for efficiency, since the rules specify the order in which to search antecedents. By default, the depth of the search tree is limited to twenty, though this can be adjusted through advice wrappers. **`Query`** is built on top of **`ask`** and supports the same parameters, advice wrappers, and context restrictions.

The Logic-based Truth Maintenance System (LTMS) actually has three main functions: it caches results in RAM to reduce KB accesses, it provides an audit trail for explanation, and it maintains consistency when assumptions change. Every time a fact is retrieved from the knowledge base, the fact is stored in working memory along with a justification that indicates that it came from the KB. Facts that are inferred are justified with the kind of inference and any antecedents that support them. When belief in an antecedent changes, the change is rapidly propagated without re-computing the actual inference. This is especially important for higher-level reasoning such as planning, where many states are projected and retracted as it searches for a solution.

The microtheory contexts partition the knowledge base into collections of locally consistent and related assertions (Lenat, 1998). Every query in FIRE is made with respect to a context that limits KB retrievals to only those contexts accessible from the query context (via microtheory inheritance). This has the effect of both improving efficiency by reducing retrieval results and ensuring correctness by avoiding retrievals from unintended contexts (such as hypothetical, fictional, or counterfactual microtheories). In earlier versions of FIRE, before contexts were strictly enforced, there was always the possibility of seeing statements about Frodo Baggins in inferences involving military courses of action, for example.

Outsourced predicates are the mechanism that makes FIRE a *federated* reasoning engine. Inferences that are implemented via external code are reified through predicates defined in the knowledge base. When a query is invoked that involves an outsourced predicate, FIRE invokes the associated code rather than attempting to retrieve an answer from the KB. Often, outsourced predicates serve as an interface to an external package, such as a simulation, game, or analogy system that can be thought of as an alternative knowledge source. For example, analogical matching is carried out via the Structure-Mapping Engine (Falkenhainer et al 1989) via outsourced predicates. Similarly, analogical retrieval and generalization are both invoked through predicates that ask for reminding and/or add cases to case libraries or generalization contexts. Other times, an outsourced predicate may simply be a more efficient way to encode an

algorithm. One benefit of invoking code in this way is that declarative and procedural knowledge remain distinct and rules do not textually embed pieces of code in them.

In order to provide declarative control over reflexive reasoning, FIRE uses advice wrappers that encapsulate queries. Although the control strategy itself is fixed in reflexive reasoning, wrappers can specify bounds on resources, contexts, fact sources, and recursive depth of inferences. For example, `(wmOnly <query>)` limits `<query>` to be solved using only the contents of working memory, without accessing the knowledge base at all. `(withBackchainingDepth <n> <query>)` puts a strict bound on recursive backchaining. `(localOnly <query>)` limits lookups to the current microtheory environment without inheritance. All told, there are about 25 such wrapper predicates. Over time, we have come to rely less on the strict stratification of **ask** and **query** and have adopted a more nuanced control over resources via the advice wrappers.

Deliberative Reasoning

The deliberative reasoning services are designed to support simple kinds of reflective control. Currently, there are two services at this level: **`solve`** and **`plan`**.

`Solve` is, at its core, an And/Or graph search algorithm (Forbus & de Kleer, 1993). By itself, this is not deliberative. The deliberative aspect derives from three features:

1) A *suggestions* architecture that explicitly represents alternative solution strategies for problems in the form of and/or graphs,
2) *preferences* between suggestions, implemented via estimated costs for particular nodes and explicit preference rules, and
3) *reification* of the graph in a way that permits incremental operation, reasoning about the state of the search, halting, or requesting additional answers.

Unlike the reflexive inferences, **`solve`** searches in a best-first manner, governed by cost estimates calculated by the system, and produces single solution by default. It can incrementally produce additional solutions on demand and can be interrupted and resumed by external systems. The explicit representation of suggestions and preferences is designed to permit reflective control and make it possible to learn heuristics for problem solving.

`Plan` is a Hierarchical Task Network (HTN) planner modeled after the SHOP algorithm (Nau et al 1999). It reduces high-level, non-operational tasks to linear sequences of ground primitive actions that can be executed. As with **solve**, it is not the planning algorithm itself that is deliberative, but the heuristics for preferring one task expansion over another and facilities for incremental operation.

The planner and solver are not built on top of each other, but are complementary services. **Solve** is a purely inferential mechanism that searches for sets of bindings to answer queries. Although planning is also inferential, the execution of plans provides a way to take action in the world. An agent can thus execute actions that generate new knowledge and write it out to the knowledge base, or execute actions in an external system such as a game.

Another deliberative aspect of the planner is the ability to generate partial plans that include actions to defer further planning until execution time. The `doPlan` operator invokes planning as a primitive action. This allows the plan/execution system to take actions that generate information and then resume planning given that new information. The `doSolve` operator can likewise invoke the solver as a primitive action. The execution system that makes this incremental behavior possible is not part of FIRE itself, but is implemented as part of the Companions architecture that builds upon it. We describe Companions next.

Example FIRE application: Companions

The Companions cognitive architecture is exploring the idea that analogical processing is a central operation in human intelligence. Because we believe that common sense relies on experience, and to date the more social creatures are, the smarter they tend to be (Tomasello, 1999), Companions are being built with the goal of being social organisms, capable of interacting with people and thereby learning from them, in an apprenticeship fashion (Vygotsky, 1962). For engineering reasons, Companions are implemented via a distributed agent architecture. This enables us to escape the limitations of a single desktop machine by distributing agents over a cluster nodes. However, multiple agents can also be run on a single machine, for portability and ease of experimentation. Depending on configuration, agents either share a KB running on a separate KB server, or have their own local copy of the KB, for efficiency. A journaling mechanism is used to synchronize KB contents across agents.

The agents that make up a Companion communicate via KQML (Labrou & Finin, 1997). With the exception of a Facilitator that sets up communication links between other agents, and a Session Manager that provides a GUI for starting, stopping, and debugging, agents are basically a lightweight layer of communication services around FIRE. Most of the operations of an agent are specified either via HTN plans or solve suggestions, with simple "heartbeat" computations invoking planning or solving as necessary.

Companions have been used in a variety of experiments. In modeling everyday physical reasoning (Klenk et al 2005) and transfer learning in AP Physics (Klenk & Forbus, 2009), **solve** was used with cases retrieved via analogy to learn by accumulating examples. Transfer learning in general game playing (Hinrichs & Forbus 2009) used **plan** to conduct experiments to learn winning strategies for games, and SME for analogical mapping to transfer learned strategies to new games. SAGE has been used to model learning of naïve conceptions of motion from sketched comic strips (Friedman & Forbus, 2009) and sketched concepts using near-misses as well as

generalization (McClure et al 2010). These experiments would not have been possible without FIRE.

Related Work

Many of our design choices are inspired by Cycorp's Cyc family of reasoning engines. This includes the use of specialized storage for common kinds of facts, hidden behind a standard pattern-directed retrieval interface for uniform access, the use of microtheories to provide a logical environment for reasoning, and the use of a variety of procedural attachments for efficient inference. We diverge from them in making analogical reasoning primary in the reasoning system – analogical matching and retrieval, for example, are more primitive than backchaining, since they occur using **ask**.

Discussion

We believe that the key to common sense reasoning is analogical reasoning and learning over vast amounts of experience. We have described FIRE, the KB and reasoning infrastructure we have built to experiment with experience-based systems. Rapid retrieval from the KB while respecting logical environment constraints is supported by mention integers and a persistent TMS. Specialized support for case libraries and generalization contexts support analogical reasoning and learning. Stratified reasoning services provide a combination of reflexive and deliberative inference, supporting constrained logical inference, traditional problem solving, and planning. These techniques are used in the Companions cognitive architecture, and are key to its performance in a variety of successful experiments.

There are several avenues for future work. First, we are planning larger-scale experiments with Companions, with the goal of improving the architecture and our NLU and sketch understanding capabilities to the point where they are able to learn as apprentices and learn by reading. Second, a problem with the current strategies in the reflexive layer is that they return all solutions by default. This does not scale well, and we plan to explore techniques like lazy retrieval or spreading-activation memories (cf. Anderson 2007) to find better solutions.

Acknowledgements

This research is supported by the Defense Advanced Research Projects Agency and by the Office of Naval Research.

References

Anderson, J. 2007. *How Can the Human Mind Occur in the Physical Universe?* Oxford.

de Kleer, J. An assumption-based TMS, Artificial Intelligence 28(2):127-162,1986.

de Kleer, J. A hybrid truth maintenance system, PARC Technical Report, January, 1992

Falkenhainer, B., Forbus, K., Gentner, D. The Structure-Mapping Engine: Algorithm and examples. *Artificial Intelligence*, 41, 1989, pp 1-63.

Forbus, K. and de Kleer, J., *Building Problem Solvers*, MIT Press, 1993.

Forbus, K., Klenk, M., and Hinrichs, T. , 2009. Companion Cognitive Systems: Design Goals and Lessons Learned So Far. *IEEE Intelligent Systems*, vol. 24, no. 4, pp. 36-46, July/August.

Friedman, S. and Forbus, K. 2009. Learning Naïve Physics Models and Misconceptions. *Proceedings of CogSci-09.*

Hinrichs T. and Forbus K. (2009). Learning Game Strategies by Experimentation In *Proceedings of the IJCAI 2009 Workshop on Learning Structural Knowledge From Observations*, Pasadena, CA.

Klenk, K., Forbus, K., Tomai, E., Kim, H., and Kyckelhahn, B. 2005. Solving everyday physical reasoning problems by analogy using sketches. *Proceedings of AAAI-05*.

Klenk, M. and Forbus, K. 2009. Analogical model formulation for transfer learning in AP Physics. *Artificial Intelligence* http://dx.doi.org/10.1016/j.artint.2009.09.003

Labrou, Y., and Finin, T. 1997. A proposal for a new KQML specification. TR CS-97-03, CS & EE Department, University of Maryland Baltimore County.

Lenat, D. 1998. The Dimensions of Context-Space. Cycorp Technical Report, October.

Lockwood, K. & Forbus, K. 2009. Multimodal knowledge capture from text and diagrams. *Proceedings of KCAP-2009*.

McDermott, D. 1975. Very large PLANNER-type databases. AI Memo 339, MIT AI Lab. http://hdl.handle.net/1721.1/6240

McLure, M., Friedman, S., and Forbus, K. 2010. Learning concepts from sketches via analogical generalization and near-misses. *Proceedings of CogSci10.*

Mostek, T., Forbus, K. and Meverden, C. 2000. Dynamic case creation and expansion for analogical reasoning. Proceedings of *AAAI-2000*. Austin, Texas.

Dana S. Nau, Yue Cao, Amnon Lotem, and Hector Muñoz-Avila. SHOP: Simple hierarchical ordered planner. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 968-973, 1999.

Tomasello, M. 1999. *The Cultural Origins of Human Cognition*. Harvard University Press.

Vygotsky, L. (1962) *Thought and Language*. MIT Press.