Using Reasoning Threads in Enhanced Semantic Networks

Richard Cardone, Rangachari Anand, Xuan Liu, Leora Morgenstern, Erik Mueller, Doug Riecken
IBM Watson Research
Hawthorne, New York, USA
{richcar, ranand, xuanliu, leora, etm, riecken}
@us.ibm.com

Calvin Lin
Department of Computer Sciences
University of Texas at Austin
Austin, Texas, USA
lin@cs.utexas.edu

ABSTRACT

A fundamental problem limiting the use of knowledge based systems is the difficulty of managing and evolving knowledge over time. As the amount of knowledge in a knowledge base (KB) increases, the ability to predict how any change will affect system behavior decreases. Moreover, because of the highly interconnected nature of knowledge, KB complexity can grow exponentially with the number of facts and rules. We address this scalability problem by leveraging *enhanced semantic networks (ESN's)* and the idea of *reasoning threads*. We describe the design, implementation, and capabilities of our ESN system, the *Semantic Engine*, and show how to build useful knowledge base systems with it. We also present novel static and dynamic analysis techniques that aid in knowledge maintenance. Our static analysis can detect anomalies, such as output states that can never be reached, and can be used to compare the before and after effects of a KB change. Our dynamic analysis provides immediate feedback to guide knowledge engineers in an interactive authoring environment.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.4. Software/Program Verification. I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods

Keywords

Semantic network, knowledge base verification.

1. INTRODUCTION

Knowledge based systems (KBS's) model human knowledge and reasoning to address problems that are not easily solved using deterministic algorithms. These systems usually apply some type of search strategy to problems that can have zero or more solutions. For example, a simplistic implementation of a production rule system starts with a set of facts and sequentially determines if those facts satisfy the antecedents of each of the system's rules. When a rule is satisfied, its consequent can add new facts to working memory, which spawns a new iteration of the reasoning process. The order in which the rules are visited and the facts are added constitute a search strategy; the search space generated by such reasoning can quickly become intractable as a knowledge base grows. Algorithms such as RETE [6] and its successors have been developed to make such processing more efficient.

The important point, however, is that this computational complexity not only makes KBS reasoning difficult for computers to execute, but it also makes it difficult for humans to understand. Indeed, a crucial question in knowledge base (KB) management is, "How will a particular change to a KB affect system behavior?" The answer to this question depends on understanding how reasoning interacts with knowledge and on the availability of tools that help developers analyze that interaction.

KBS's typically use some variant of first-order logic to reason about their knowledge. The knowledge itself can have many possible representations, including rule bases, semantic networks, frames [9], and description logics [12]. As the representation becomes more expressive, reasoning becomes computationally more complex, and system behavior becomes harder for developers to predict [3].

We argue that a KBS can execute in a simplified, understandable way and still do useful work. This paper introduces the *Semantic Engine (SE)*, a knowledge base system that uses *enhanced semantic networks (ESN's)* and a simplified approach to reasoning. This simplified approach allows us to define *reasoning threads*, which are the valid paths through the ESN that our reasoner explores. By encapsulating knowledge in a localized representation, reasoning threads allow our system to execute quickly and allow us to perform analyses that facilitate knowledge creation and evolution. We present a novel static analysis framework that detects anomalies, interesting characteristics, and the effects of KB changes. We use dynamic analysis to guide knowledge engineers as they create and modify KB content. Developers use these analysis tools to understand and verify the behavior of KBS's.

This paper makes the following contributions:

- We show how ESN's simplify KB reasoning.
- We present novel static analysis techniques for verification of ESN systems.

- We present new dynamic analysis techniques that interactively guide ESN evolution.
- We describe how to build useful expert systems with the Semantic Engine.

The larger view is that KBS's face the same life-cycle challenges as conventional software systems, as well as a few additional challenges. Thus, one goal of this paper is to draw attention in the software engineering community to the unique issues and opportunities presented by KBS's.

Section 2 provides background on KBS software engineering issues. Section 3 discusses SE's main concepts and provides background for the static and dynamic analyses sections, Sections 4 and 5, respectively. Section 6 describes SE applications. Section 7 discusses SE's strengths, weaknesses, and future work. We conclude with related work and final remarks.

2. BACKGROUND

The key architectural components of a KBS are (1) a *knowledge base* that contains a collection of related facts and rules, (2) a *reasoning engine* (or *reasoner*) that uses the knowledge base to draw conclusions and prove new facts, and (3) an application program that invokes the reasoning engine for specific inquiries. This tripartite architecture separates various concerns, which should simplify system maintenance, comprehensibility, and evolution [15]. The modularity of the KBS architecture mirrors that of database systems. Specifically, the KB's role is to store knowledge; the reasoner's role is to execute logic over that stored knowledge; and the application's role is to query the knowledge.

The analogy to database systems, however, breaks down when we recognize that KBs do not contain passive facts so much as the rules by which new facts can be derived, and that reasoners are not so much retrieval engines as they are logic-based search engines. Reasoners implement some form of mathematical logic in order to prove or disprove conjectures in the knowledge domain. As the number of rules in a KB increases, the interconnections between those rules—between the consequents and antecedents of different rules—can grow exponentially and can lead to search spaces too large for reasoners to exhaustively inspect.

From a software engineering point of view, KBS development presents all the life-cycle challenges of conventional software plus unique challenges associated with KBs and reasoners. In particular, extra consideration must be given to knowledge ***acquisition***, ***encoding***, ***verification***, and ***validation***. In addition, the ability for humans to understand and predict system behavior is important for knowledge ***evolution***. The remainder of this section discusses how each of these considerations impacts the development of correct and reliable KBS's.

To populate a KB, knowledge is acquired from domain experts and then encoded into the KB's knowledge representation. The pitfalls in acquiring domain knowledge from experts are well-documented and include restrictions on the availability of experts, the difficulty experts can have in explaining their knowledge, and the fact that experts often disagree with each other [22]. Knowledge encoding is also error prone; checking it can require time-consuming coordination between domain experts and knowledge engineers.

Before a KBS is deployed, it is verified to determine if it executes as specified and it is validated to determine if it meets user requirements. Verification can be thought of as a prerequisite of validation, since a system that does not conform to specification is unlikely to meet user requirements. For correct and efficient reasoning, a KB needs to be consistent, complete, and devoid of circularities and redundancies. These terms have different meanings for different knowledge representations, but, in general, a KB can be statically analyzed to verify its structure and detect *anomalies*. Anomalies do not necessarily indicate KB errors, but instead report to the knowledge engineer content that might warrant closer examination.

Anomaly detection is either domain dependent or independent. Domain dependent approaches use metadata to constrain KB content. Since this metadata also needs to be verified and maintained, it can complicate the original problem of KB verification [13]. Domain independent verification first translates a KB into a representation suited to analysis, such as a graph, and then tests for various properties. These properties can be as simple as detecting rules that can never fire in a rule-base or detecting disconnected regions in a semantic network. Anomalies have been classified in several studies [4][13][16]; here is a summary of the Preece/Shinghal formal classification scheme [17]:

Redundancy – when rules are extraneous, subsumed, or unusable

Ambivalence – when valid input leads to contradictory results

Circularity – when reasoning from a rule leads back to that rule

Deficiency – when valid input is not used or leads to no output

Whether the above conditions represent actual KB flaws often depends on the reasoning procedure used. For example, the SE reasoner handles cases of circularity without generating a runtime error. Thus, the detection of circularity during analysis would only be informational in SE.

KB validation is concerned with the reliable generation of appropriate output for every possible input. A fundamental issue is that directly testing every set of input values is impractical in all but the simplest systems. Another problem is that KBS's are used in domains where there can be more than one acceptable result and part of the problem is choosing among possible results. For instance, an expert system might be considered reliable if it always agrees with experts in simple cases and if it agrees with at least some experts in complex cases. A common theme in KB validation is finding small yet representative sets of test cases that allow one to confidently predict system behavior [13].

KBS's typically address problems that involve competing goals, understanding context, or using common sense—in a word, problems that demand *judgment*. For example, KB technology is used to diagnose medical conditions [21], render legal advice [23], and configure complicated systems [1]. In general, KB's need to change as quickly the domains they model. But when a KB is modified, how can one guarantee that the KB is still consistent, that the quality of its output hasn't deteriorated, or that the system will behave as intended? Consider the problem of validating changes to the XCON system, which by 1988 contained over 10,000 rules for configuring DEC computers. Then consider that approximately 40% of the rules were changed, added, or deleted each year [1].

The ability to safely evolve a KB is crucial in justifying a KBS investment.

Our main motivation is to increase the use of KBS's by simplifying KB management. In particular, we want to build expert systems using a knowledge representation that is comprehensible to non-specialists. Our approach uses semantic networks that can be graphically displayed and that behave operationally like decision trees or even linear reasoning threads. This simple operational model is more accessible to non-technical domain experts [23] and can improve knowledge acquisition and encoding. Our simplified approach should also make verification and validation easier, as well as benefit explanation generation and other natural language processing that expert systems often perform.

3. THE SEMANTIC ENGINE (SE)

3.1 Enhanced Semantic Networks

A *semantic network (SN)* is a directed graph that consists of nodes that represent concepts and edges that represent relations between concepts. SN's are appealing because their graphical structure can help us visualize reasoning. Intuitively, paths in an SN should correspond to reasoning chains. Upon closer examination, however, arbitrary paths in a SN do not generally correspond to valid reasoning [24].

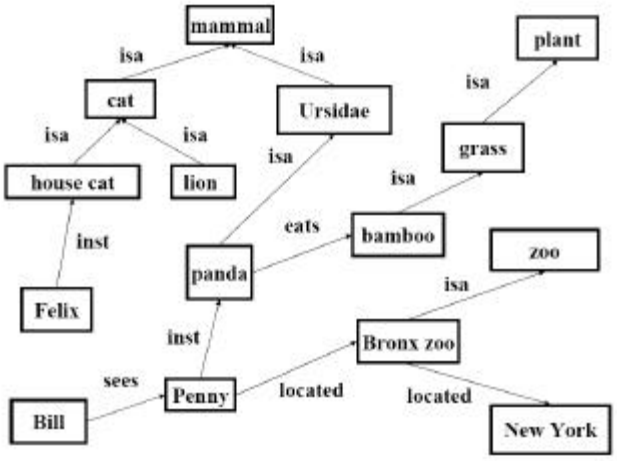

Figure 1: Only some SN paths correspond to valid reasoning

Consider Figure 1, which depicts a SN that represents knowledge about the animal kingdom and Bill's visit to the Bronx Zoo. We can assign meaning to many paths. For example, the path from `Penny` to `mammal` intuitively means that "Penny is a mammal." Many paths, however, do not have an obvious meaning: What can we conclude about the path from `Bill` to `plant`? That Bill is a plant? That Bill eats some plants, all plants, or only plants? In other cases, it's not clear how we draw a correct conclusion: When traversing the path from `Bill` to `New York`, how can we conclude that Bill is located in New York? In general, arbitrary SN's have no clear semantics and no clear specification of valid reasoning.

One way to force all paths in a SN to be meaningful is to limit the expressivity of the network. For example, inheritance networks build meaningful taxonomies by limiting their relations to subtype (*isa*) and membership (*inst*). This approach is restrictive, however, since one can only reason about whether something is a member of a class or whether one class is a subset of another.

In SE, we propose a solution that lies between loosely defined reasoning and limited, single-mode reasoning. The key idea is to recognize that there are specific patterns in a SN that correspond to valid reasoning. We identify valid reasoning paths in a SN by determining whether a path fits any of a predetermined set of reasoning patterns. We define valid reasoning in a SN by means of these fixed patterns, which we encode as regular expressions. The resulting *enhanced semantic network* (ESN) extends the expressiveness of inheritance networks while maintaining a precise definition of valid reasoning.

An ESN is a semantic network in which each node is specified with a *node type* and each link (edge) with a *link type*. Node types and link types define two disjoint, finite sets in an ESN. Each ESN is also associated with a list of valid reasoning patterns, which are specified as regular expressions (*regexes*) composed using node types and link types.

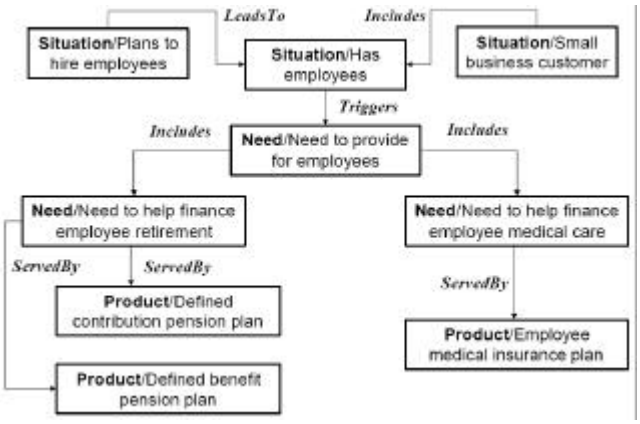

Figure 2: Small business banking ESN

Figure 2 shows an ESN fragment that models banking services for small businesses. Each node is labeled with its type and its name; each link is labeled with its type. The regex below, which we will call the *Recommend regex*, defines reasoning paths that begin at Situation nodes and end at Product nodes.

Situation ((LeadsTo Situation) | (Includes Situation)) Triggers Need (Includes Need)* ServedBy Product*

In general, a regex starts with one or more node types and is followed by zero or more link/node type sequences. Any number of regexes can be defined on an ESN. For example, the regex, *Product -ServedBy Need*, can be used to discover the immediate Needs served by a Product (the minus sign specifies backwards traversal on *ServedBy* links).

To further restrict traversal, ESN's also support formulas on nodes and links. For instance, the Recommend regex above generates nine paths, one that connects each Situation to each Product. If the following formula appears on the "Employee medical insurance plan" node, then traversal would only proceed to that node for businesses with more than five employees.

NumberOfEmployees(Customer) > 5

ESN formulas are formulas in the first-order logic sense: Boolean expressions that contain arithmetic, relational, and logical operators. The supported data types are string, number, Boolean, and user-defined enumerations. The value of an enumerated type can be existentially or universally quantified in a formula. Formulas can also reference strongly typed, user-defined functions. These functions take zero or more parameters and return a supported type. Functions that return Boolean can be thought of as predicates in first order logic. In the formula above, the function *NumberOfEmployees* takes a parameter of type *Person* (assume *Customer* is an enumerated value of *Person*) and returns a number.

3.2 SE Process Model

In this section, we discuss how SE reasons with ESN's. A formal treatment of ESN reasoning demonstrates that SE reasoning is equivalent to a subset of first-order logic [10].

On input, the SE reasoner takes an ESN, a regex, an optional start node list, and an optional profile. The profile defines a set of read-only facts that are used to evaluate node and link formulas. These facts are specified as assignments to grounded functions (i.e., functions with no variables). For example, the following fact would cause the formula presented in the last section to evaluate to true: *NumberOfEmployees(Customer)=20*.

Start nodes are nodes where the reasoner begins regex matching. If start nodes are provided as input, they are the only nodes used to begin matching. Otherwise, the reasoner calculates the set of start nodes before proceeding. Start nodes are calculated using a specially designated formula, the start condition formula (or *sformula*), which can be defined on nodes. When generating the start node set, the reasoner uses profile facts to evaluate all sformulas in the ESN. The nodes whose sformulas evaluate to true are added to the start node set.

During reasoning, the SE reasoner searches for all *valid paths* in the ESN. Valid paths are those that match a regex. Regex matching corresponds to constructing a string from a path's node and link types and determining if the regex accepts that string. Paths begin at start nodes; each time a regex match occurs, the reasoner checks for an optional precondition formula (*pformula*) before advancing. These formulas, introduced in the last section, must evaluate to true for traversal to continue along a path. Reasoning ends when traversal has been attempted on all valid paths.

On output, the SE reasoner reports results in a *reasoning graph*. Included in the reasoning graph are the *output nodes*, which represent accepting states in the regex; the *reasoning threads*, which are valid paths whose formulas all evaluate to true; and a list of *unknown atoms*, which are grounded functions[1] whose values are not known. During traversal, if the reasoner encounters a function whose value is unknown, the reasoner adds the function to the unknown atoms list and aborts the reasoning path.

SE reasoning reduces computational complexity and increases overall comprehensibility in several ways. First, by specifying valid reasoning, regexes eliminate most network paths from consideration. Also, SE profiles act as read-only working memory during reasoning, so the order in which paths are processed does not affect the final reasoning result. Moreover, reasoning thread traversal is idempotent and independent of other traversals.

An ESN corresponds to a set of well-formed formulas in first-order logic and SE reasoning corresponds to simple theorem proving over those formulas [10]. Indeed, ESN's provide a remarkably compact representation for rules. One can view any legal path in an ESN as being analogous to a rule in a rule-based system, and a small ESN can yield a large number of legal paths. This compact representation is one of the reasons KB's have been relatively small in the applications that we have written (see Section 6). Compactness is also due in part to a concise and powerful ontology that we have developed [11].

SE reasoning also provides expert systems with a framework for a conversational interaction with users. The reasoner delivers its conclusions or recommendations as output nodes. The reasoner also indicates that reasoning could be extended if more information were known by returning unknown atoms. An application can use these unknown atoms to query users for more information and then invoke the reasoner with an updated profile. SE provides a natural language processor that generates English questions from unknown atoms and generates explanations from reasoning threads. This language support improves user experience with little effort from application developers.

3.3 SE Architecture

Figure 3 below shows the software stack for SE applications and for the SE authoring environment. The SE Authoring Workbench consists of editors, which include graphical and text-based tools, and analyzers, which we discuss in detail in the following sections. The authoring modules interact directly with the KB and with the SE Runtime.

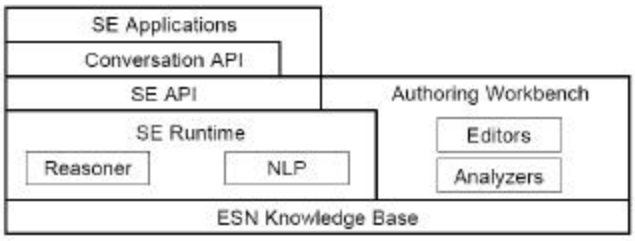

Figure 3: SE Runtime and Design-Time Architecture

The main components of the SE Runtime environment are a reasoning engine and a natural language processor (NLP). Both components interface with the KB. We described the reasoner in the previous section. The NLP dynamically generates explanations from reasoning threads, questions from unknown atoms, and question justifications from unknown atoms and partial paths. The NLP uses various heuristics and tuning parameters to generate concise, non-repetitive, natural sounding, English text that can be presented to users by an application. We do not discuss language processing in depth in this paper.

The SE API provides the public interface to the SE Runtime. Applications use the SE API to invoke the reasoner and the language processor. The Conversation API supports conversational

[1] To ground a formula, existentially quantified variables are replaced with a disjunction of enumerated values and universally quantified variables are replaced with a conjunction of enumerated values.

control flow in interactive applications. The main function of this layer is to support one or more concurrent dialogues between an expert system and its users. Each dialogue consists of currently known facts, questions posed by the system, and recommendations offered by the system on various user-selected topics.

SE is implemented in Java under Eclipse. The authoring environment is integrated into the Eclipse workbench as plug-ins; the SE runtime code executes under Eclipse or as standalone Java code. All the above components have been implemented, though the Conversation API design is still evolving. In Section 6, we discuss our experience building several prototype systems and one production system using SE.

4. STATIC ANALYSIS

SE uses both declarative and procedural information to statically analyze a KB. The declarative information consists of KB nodes, links, formulas, functions, regexes, and types. The procedural information consists of valid paths calculated by the reasoner. In practice, we can usually enumerate all valid paths during analysis because of the filtering efficiency of regexes. For example, in an ESN application that contains over 600 nodes and 900 links, the main regex generates only 21 valid paths. This ability to statically inspect all relevant paths means that we can perform analyses that would not be practical in other systems.

4.1 How Static Analysis Works

The basic idea behind our analysis is to calculate the facts required to traverse any node or link in any reasoning thread. This idea, inspired by compiler dataflow analysis, provides the raw data used to characterize KB structure and to find KB anomalies. We begin by describing how such data are calculated.

Let *P* be a valid path induced by a regex in some KB and let *A* be a node or link in *P*. We define a *model*, *M(A)*, to be a set of facts that satisfies the precondition formula at *A*, the precondition formulas for all nodes and links that precede *A* in *P*, and the start condition of *P's* start node. A fact is a ground atom of the form, *<functionSymbol>(groundedArgs)=<constant>*, such as *Age(Man)=33*, *IsMarried(Man)=True*, or *Spouse(Man)='Mary'*. When the reasoner traverses a path, empty precondition formulas are implicitly satisfied; otherwise, the reasoner uses facts to determine if it can proceed. This traversal can be thought of as computing a model at each point in the path.

We statically calculate models by beginning at a start node and assuming whatever facts are necessary for traversal to continue on the path. We convert our calculation into a *formula satisfiability (SAT)* problem by encoding SE formulas as formulas of Boolean variables in propositional calculus. Since *A* can have zero or more models, we can only say what possible sets of facts *could be* in effect at runtime. If *A* has no models, then we know the reasoner cannot traverse that point in the path.

Given (*i*) a KB, (*ii*) one or more regexes, (*iii*) an optional input node set, and (*iv*) an optional initial fact set, our goal is to statically calculate all models at all points in the valid paths. Our algorithm performs the following steps:

1. Generate all valid paths that conform to the regexes.
2. Prepare formulas for CNF conversion.
3. Generate constraints and complete CNF conversion.
4. Encode CNF formulas into DIMACS format.
5. Use a SAT solver to find all models.

Step 1 generates valid paths by running the reasoner while ignoring all formulas. If multiple regexes are specified, we invoke the reasoner with each regex and accumulate the resulting paths. For this discussion, we assume the common case of a single regex.

Step 2 prepares formulas for conversion to conjunctive normal form (CNF: conjunction of disjunctions) and, ultimately, to the standard DIMACS [5] format for SAT solver input. For each formula on a path, we perform algebraic and syntactic simplification; we ground quantified formulas; we substitute values from initial facts where possible; and we replace cardinality, implication, and equivalence expressions with conjunctive and disjunctive expressions. Formulas can only be quantified over enumerated types, so we have the closed world assumption that always permits grounding. Cardinality expressions, which have the form $min\ \{f_1, f_2, \ldots, f_n\}\ max$, specify a minimum and maximum number of Boolean formulas, f_i, $0 < i = n$, that must evaluate to true. Cardinality expressions are converted to a disjunction of conjunctions.

Step 3 preserves semantics when converting ESN formulas into propositional formulas. Conversions involving (1) equality expressions with literals, (2) range expressions with literals, and (3) multiple atom expressions require special consideration.

Consider the equality expression $f(args)=L_1$, where *f(args)* is a ground atom and L_1 is a literal value. We can represent $f(args)=L_1$ with the propositional variable *v1* and represent the negated expression, $f(args)?L_1$, with *-v1*. Similarly, variable *v2* can represent $f(args)=L_2$, $L_1?L_2$. If both equality expressions appear in formulas, then we insert the clause (*-v1 | -v2*) wherever *v1* appears to indicate that *v1* implies *-v2*. We do a similar insertion wherever *v2* appears. These insertions preserve the original ESN semantics where *f(args)* can equal exactly one value $L_1, L_2, \ldots, L_n$, where $L_i?L_j$ if $i ? j$. We generalize this *constraint insertion algorithm* to handle both equalities and inequalities.

We also introduce *numeric ranges* to encode relational expressions like *f(args)>5*. This encoding replaces a literal numeric value in a relational expression with a *range name* in an equality expression. For instance, *f(args)>5* would be transformed into *f(args)=#gt5*. Next, we enhance the above constraint insertion algorithm to recognize range names and to generate appropriate constraints for incompatible assignments. Consider the following two relational expressions with their range name substitutions and SAT variable assignments:

f(args)>5 ➔ *f(args)=#gt5* ➔ *v1*
f(args)<0 ➔ *f(args)=#lt0* ➔ *v2*

Since the values in ranges *#gt5* and *#lt0* are mutually exclusive, we insert the constraint (*-v1 | -v2*) as previously described.

Certain expressions, however, cannot easily be assigned propositional variables. For example, non-linear algebraic expressions, such as *f(args)^3=5*, are not handled by our current algebraic simplifier. A more fundamental problem occurs when we try to assign variables to expressions containing multiple atoms, such as *Age(Husband)<Age(Wife)*. If we assign this condition to variable *v1*, when *v1* is assigned a truth value in a model, we still would

not know specific literal values for *Age(Husband)* and *Age(Wife)* that would satisfy the model. Since these atoms can be interrelated in many expressions, determining their values might require solving a difficult constraint satisfaction problem. The same problem occurs when the value of a nested atom is unknown, such as *ModelOf(Car)* in *IsFast(ModelOf(Car))*.

To address these problems, we detect complex algebraic and multiple atom expressions and then ask the user to provide specific additional facts. When these facts are provided, they are used to simplify expressions. Otherwise, the analysis excludes threads that contain the complex expressions. To date, and without deliberate planning, ESN applications contain few or no complex expressions, which might indicate that these expressions are atypical in such applications.

Once complex expressions have been processed and constraints have been inserted, we convert ESN formulas to CNF. To avoid an exponential increase in the size of formulas, we use the conversion algorithm specified by Giunchiglia and Sebastiani [8].

Steps 4 and 5 involve the straightforward translation of CNF formulas to DIMACS format and the invocation of a SAT solver. An analysis run can invoke the SAT solver once for each formula on each valid path. Whenever possible, we cache intermediate results to speed up processing. For example, since formulas are cumulative in a thread, we append the current formula encoding to the previous DIMACS encoding as we progress through a thread. We also cache formula conversion data between analyzer runs.

In current applications, SE achieves sub-second analysis execution times on an IBM T40 laptop using the Java-based SAT4J solver [18]. These executions often include 70 or more invocations of the solver. The number of solver invocations is at most the number of formulas on all valid paths, which is roughly proportional to the size of the KB.

4.2 How Static Analysis is Used

SE static analysis calculates all possible models at each node and link in a valid path. These models are minimal in that only atoms that appear at or before a location in a path are assigned truth values at that location. If the last node in a path has at least one model, then the path is a reasoning thread. Each model at a thread's terminal node defines a set of facts that allows that thread to be traversed. Paths that cannot reach their terminal nodes are reported as *dead ends*, a type of ambivalence anomaly that might indicate a KB error. In fact, we have detected and removed such paths from our KB's using this analysis.

The SE analyzer optionally generates coverage tests by writing new profiles to disk. Each profile contains a set of facts that allows at least one thread identified during analysis to be traversed by the reasoner. Taken as a group, the profiles exercise all threads. The current generator uses a greedy algorithm to create the minimal number of profiles for a given set of threads, but other algorithms are possible. SE also detects disconnected regions in KB's using a static analysis that does not rely on threads.

Our ongoing work focuses on different ways to characterize a KB. *Steady state* analysis allows a knowledge engineer to probe a KB and understand its structure. *Change impact* analysis compares the characteristics of a KB before and after changes are made. We can think of these analyses in terms of the questions that they answer about a KB. These questions can be categorized into three levels. At the regex level, we ask questions about the reasoning threads induced by one or more regexes. At the thread level, we ask questions about node and link traversal. At the fact level, we ask questions about the facts asserted at various points during reasoning.

4.2.1 Steady State Analysis

Steady state analysis involves a single execution of the analyzer. The reasoning graph returned by the analyzer contains the raw data needed to answer the following questions.

Regex level questions: What nodes are not included in any thread? Do two or more regexes share any nodes? Are the threads induced by a regex confined to a specific region of the KB? The answers to these questions allow us to split a KB along regex lines or to eliminate unused nodes and links.

Thread level questions: How many ways are there to reach a node, especially an output node, for a given set of regexes? How many threads start at a particular node? Is there a thread that connects start node *S* to some other node *N*? What nodes *dominate* other nodes, where node *D* dominates node *N* iff *D* is traversed before *N* on every thread in which *N* appears? The answers to these questions can help us predict reasoner behavior and validate the KB domain model.

Fact level questions: What facts are asserted at a node across all threads that traverse the node? What output nodes are reached when fact *F* is asserted? Or when *not F* is asserted? Given fact *F*, is there a fact *F'* such that *F'* is always asserted when *F* is asserted? The answers to these questions use the analyzer-generated models to explain how facts guide reasoning.

4.2.2 Change Impact Analysis

Change impact analysis involves two executions of the analyzer, one that occurs before KB changes are made and one that occurs after. The basic idea is to compare the results of the two executions to understand the affect KB changes have on system behavior. By comparing the answers to the above questions both before and after changes are made, we should be able to identify unintended consequences caused by the changes. For example, we can determine if the number of threads has changed; if the number of threads passing through a particular node has changed; if the facts asserted at a node have changed; or if the nodes and links in a thread have changed. If any of these changes are unexpected, the knowledge engineer can investigate further.

4.2.3 Static Analysis Framework

The SE analyzer provides a wealth of information that knowledge engineers can use to predict KBS behavior. The distinguishing feature of our analysis is its ability to abstractly evaluate all reasoning threads. The challenge is to allow engineers to quickly access the most useful data generated by that evaluation. For example, if an engineer wants to verify that all nodes of a certain type are output nodes, the output node list in the reasoning graph result can be consulted. It would be more convenient, however, if this type of deficiency anomaly were automatically checked and presented to the engineer. Similarly, multiple threads that have the same start

and output nodes should be flagged as redundancy anomalies without requiring a visual inspection of all threads.

Currently, SE provides the framework and raw analysis data by which many aspects of a KBS can be verified and validated. Our continuing work includes determining what information is most useful and how to present it most effectively. Once the models have been generated on each valid path, the computations described in Sections 4.2.1 and 4.2.2 are, with one exception, linear with regard to the number of nodes, threads, facts, or models. Finding the facts that are always asserted when fact *F* is asserted has complexity $O(n^2)$, where *n* is the number of facts.

5. DYNAMIC ANALYSIS

In SE, *dynamic analysis* involves the use of the reasoner during KB authoring. In this section, we discuss two ways that reasoning is integrated with authoring:

- Regression testing: We incorporate KB regression testing in the SE authoring workbench (AWB) to provide the knowledge engineer with immediate feedback after changing a KB.
- Reasoning driven authoring: The reasoning algorithm is used to suggest possible extensions for selected reasoning threads.

5.1 Regression Testing

The need for regression testing in software development is well recognized [14]. The purpose of regression testing is to ensure that existing functionality is not adversely affected when software is modified. In KBS's, the management a KB becomes more complex as the KB grows. Consider a product recommendation system in which the number of possible product recommendations increases over time. At some point, a knowledge engineer cannot keep all recommendations for all products in his head. In such situations, how can a knowledge engineer know if a KB change is safe to make?

To address this problem, we allow knowledge engineers to understand the effect a change in terms of its impact on reasoning results for particular inputs. We use the term *exemplars* to refer to sets of facts that represent scenarios that a KB is expected to handle. In applications that recommend products, for example, an exemplar would typically represent a class of customers. Each exemplar is associated with a list of invariants: nodes that either must or must not be activated when the reasoner runs with the exemplar's input. For example, in an SE banking application, we might want to ensure that existing homeowners are never offered a mortgage that is intended only for first-time homeowners. The exemplar for existing homeowners would specify that nodes related to first-time homeowners should not be activated.

Whenever the KB is modified using the AWB, the reasoner automatically runs all exemplars in the background. The difference between the reasoning output and the expected output is displayed along with any information about violations of invariants. By calculating reasoning threads, the dynamic analysis subsystem shows what nodes were newly activated as a result of a change and why those nodes were activated. This is similar to the static change impact analysis discussed in Section 4.2.2, except here we automatically check for expected results during KB authoring.

5.2 Reasoning Driven Authoring

The AWB also uses reasoning threads to suggest how to extend an ESN. This procedure, called reasoning driven authoring, is summarized as follows:

- The author selects a node in a reasoning thread.
- Using the node's *prefix*, defined as the sequence of node and link types up to and including the selected node, the AWB computes the link and node type combinations that could legally follow the selected node.
- The author selects the desired extension.

One can think of reasoning driven authoring as a kind of type-ahead for ESN editing. To illustrate how this works, consider the reasoning thread shown in Figure 2 in Section 3.1 that starts with *Situation/Plans to hire employees* and ends with *Product/employee medical insurance plan*. This thread conforms to the Recommend regex, which we reproduce below:

*Situation ((LeadsTo Situation) | (Includes Situation))**
Triggers Need (Includes Need) ServedBy Product*

If a knowledge engineer wants to extend the KB at the *Situation/Has employees* node, then the AWB would compute the prefix: *(Situation/Plans to hire employees) (LeadsTo) (Situation/Has employees)*. The following link type/node type pairs could be added after this prefix to conform with the regex:

LeadsTo ➔ *Situation*

Includes ➔ *Situation*

Triggers ➔ *Need*

If an exemplar is used in combination with reasoning driven authoring, then the author is assured that a newly added node will be activated for at least for one set of inputs.

6. APPLICATIONS

Using SE, we have implemented several *conversational expert systems*, which we define as expert systems that refine their results by interacting with users. SE's conversational support is based on the reasoner's ability to report the unknown atoms that it encounters during KB traversal. An unknown atom indicates that a valid path could be traversed further if one or more facts were known. As mentioned in Section 3.3, the SE natural language processor can generate questions from unknown atoms as well as explanations of why a question is being asked. Applications can use these facilities to elicit more information from users. When provided with these additional facts, the reasoner can explore more of the network.

Fundamentally, our conversational support depends on valid reasoning paths. These valid paths allow knowledge engineers to work at the level of domain concepts and relations rather than at the level of first-order logic. This high level approach facilitates communication with users because it matches the level of abstraction that users understand.

The most significant SE application to date has been an IBM sales recommendation system. This pilot sales application was restricted to one geographic area and one aspect of IBM's service portfolio. A group of roughly a dozen telemarketing representatives used the SE system during sales calls while another group of the same size

served as a control group. Sales measurements taken over a three month period indicated that the test group outperformed the control group by over 400%. In addition, the test group increased their sales by over 400% when compared to the same period in the previous year. Though such a limited study only indicates potential, the results were enough to put the system into production and begin a larger geographic rollout.

The sales system is a J2EE web application that calls the SE public API. The application's KB contains 164 nodes, 330 links, 42 functions, and 2 regexes. The main regex generates 61 threads with an average of 5.9 nodes and 1.1 formulas per thread. The system performs well and has required little maintenance because the domain knowledge is relatively stable. Other SE applications include a system that recommends circuit board configurations for an IBM hardware division and a system that recommends small business services offered by an international bank. This latter system implements our largest KB with 627 nodes, 937 links, 166 functions, and 3 regexes. The main regex generates 21 threads with an average of 8.4 nodes and 3.2 formulas per thread. Both systems are prototype web applications.

Currently, we are building a pilot application for a telecommunications company to assist with customer cell phone upgrades. We expect this system to provide valuable feedback about the effectiveness of our analysis tools now that a significant portion of those tools are in place. In the next section, we discuss what applications make good candidates for SE.

7. DISCUSSION

A distinguishing feature of KBS's is their ability to determine control flow at runtime. A standard rule-based system, for example, determines the next rule to fire based on the current state of working memory and the rules in its KB. At development time, knowledge engineers define *if-then* rules, but they do not explicitly specify the order in which rules will fire. At runtime, after a rule's antecedent is satisfied by the state of working memory, the reasoner determines when the rule actually fires.

This approach differs significantly from that used in traditional software where most, if not all, control flow is explicitly specified at development time. KBS's often succeed where traditional systems have difficulty because KBS's do not need to prescribe execution order in advance. This feature is especially important for applications in dynamic domains that have many data interdependencies and many special cases.

One drawback of the KBS approach, however, is that the number of possible execution paths can grow exponentially with the number of rules. This growth adds indeterminism to a computation when many more paths exist than can feasibly be executed. Also, the additional computational complexity makes analyzing and predicting KBS behavior more difficult.

SE trades away some KBS flexibility by specifying valid reasoning patterns in advance. In exchange, SE gains good performance in a simplified runtime environment. SE static and dynamic analyses depend on having a manageable number of execution paths in an environment where facts are read-only. These characteristics lay the foundation for the verification, validation, and authoring techniques described in this paper.

SE is hardly unique in devising ways to reduce KBS complexity. After seven years of development and use, the XCON rule-based system mentioned in Section 2 was rewritten to reduce rule complexity and to improve KB manageability [1]. Two key components of the new system are *algorithmic methods*, which allow execution sequences to be specified at development time, and *decision methods*, which provide a way to order execution at runtime based on current state. Both methods make control flow more deterministic and, in that sense, SE takes a similar approach.

SE inhabits the design space between dynamic rule systems and static decision trees by allowing the traversal of multiple reasoning threads in a single reasoning session. Adding or modifying a regex creates a new way to reason over the same semantic network. Adding, deleting, or modifying nodes and links can also change reasoning. SE is significantly more flexible than decision trees because its execution paths are not hardcoded.

Thus, SE is well-suited for conversational expert systems that make recommendations from a finite, well-defined set of choices that is known in advance. In addition, SE is appropriate when reasoning patterns are also known in advance. Within these constraints, SE provides a highly interactive user experience.

On the other hand, SE is not appropriate for systems that optimize or prioritize their results since SE has no way to compare recommendations or enforce global constraints. SE evaluates formulas using an initial set of read-only facts; there is no concept of a mutable working memory to which facts are added during reasoning. In addition, SE is not appropriate for complex configuration problems like those addressed by XCON because enumerating all possible configurations in advance is impractical.

Currently, we are exploring ways to increase the expressiveness of SE without jeopardizing its essential simplicity. Ideas include adding global constraints; adding axioms that a theorem prover could use to expand the set of known facts; and providing a way to order and group generated questions. This last point is most important because it provides applications with greater conversational control. The Conversation API shown in Figure 3 is where we are defining this new conversational support. For the longer term, we are considering ways to modularize KB's, ways to support collaborative KB authoring, and the introduction of node and link subtypes.

8. RELATED WORK

The Preece/Shinghal anomaly classification scheme [17] discussed in Section 2 is tailored to rule-based systems, but the anomalies it defines are applicable to SE. In SE, deficiency anomalies include unreferenced functions, unused input facts, and incomplete thread coverage of potential output nodes. Dead ends are ambivalence anomalies; threads with the same start and end nodes are redundancy anomalies. Revisiting a node without advancing in a regex is a circularity anomaly.

KBS verification has been extensively studied over the past three decades. As early as 1982, an oncological decision support system, ONCOCIN, provided domain independent rule checking as a way to find KB anomalies [20].

In KB-Reducer [7], Ginsberg uses whole-KB analysis to detect all potential redundancies and contradictions (ambivalences). His approach improves upon the pair-wise rule analysis used in

ONCOCIN, which cannot guarantee to find all anomalies. KB-Reducer first calculates the dependency relationships between rules and then it generates the environments that satisfy each rule. In the worst case, the number of generated environments is exponential in the number of *findings*, which are literals used as input. Though the worst case is unlikely in practice, in KB's with 50, 150, and 370 rules, the number of generated environments was 700, 4000, and 35,000, respectively. The running time for the largest KB was 10 cpu hours on a late 1980's workstation. The use of reasoning threads allows SE to avoid such complex computation.

The Comprehensive Verifier (COVER) [16] is split into three subsystems: the integrity checker, the rule checker, and the rule-chain checker. The author notes that the basic semantic checks performed by the integrity checker, such as validating references and value assignments, are most effective in detecting errors. In SE, integrity checking is implemented as a fundamental part of the interactive authoring environment. Whenever changes are saved in a KB, all definitions, references, and value assignments are automatically checked. Similar to KB-Reducer, COVER's rule-chain checker generates all possible environments when performing redundancy and ambivalence verification. Like KB-Reducer, this analysis has worse case exponential complexity. The verification running time for a 540 rule KB on a Sun Sparc2 workstation was 3.5 hours. A later approach that preprocesses the KB reduced the running time to 10 minutes [25]. Again, by using a simplified process model, SE can verify KB's approximately two orders of magnitude quicker.

As mentioned in the Background and Discussion sections, XCON [1] was a mission-critical, OPS5 rule system used to configure DEC hardware and software. By 1988, the 59 XCON engineers and developers had put in place a new software engineering methodology, called RIME, to manage and maintain the system's large, complex, and constantly changing KB. We already discussed how, like in SE, new programming constructs were introduced to give developers more control over the evaluation order of rules. RIME also prescribes guidelines for rule creation that co-locates rules similar along one or more dimensions, such as data dependencies, actions triggered, etc. In addition, KB management is aided by a rule classification schema that allows for indexed searches of the rule-base.

Both OWL-DL [19] and SE provide a language for constructing semantic networks, and both provide algorithms for reasoning over those networks. In an important respect, however, OWL-DL is less powerful than SE. OWL-DL, like all standard description logics, permits only reasoning over *inst* (membership) and *isa* (subclass) links, while SE permits reasoning over any type of link as discussed in Section 3.1. OWL-DL provides properties that can define more general relationships, but reasoning is limited to membership and subclass calculations.

SE also permits a particular type of reasoning not expressible in OWL-DL: that of *composition*. In SE, one can represent the sentence, *(forall x) (P(x,y) & Q(y,z) → R(x,z))*. In OWL-DL, however, composition was deliberately omitted because it can make a language intractable [2]. SE avoids that problem by using reasoning threads. In every SE application that we have examined, we have performed reasoning equivalent to the following:

1. Premise: *Triggers(Situation, Need)*
2. Premise: *ServedBy(Need, Product)*
3. Rule: *Triggers(Situation, Need) & ServedBy(Need, Product) → Recommendation(Situation, Product)*
4. Inference: *Recommendation(Situation, Product)*

Composition is required to express the rule in step 3, and this is not supported by OWL-DL.

In another important respect, however, OWL-DL is more powerful than SE. Like most standard description logics, reasoning in OWL-DL includes efficient algorithms for *subsumption* (determining whether one class is a subset of another) and *classification* (determining where in a taxonomy a class belongs). There are no such tools for SE, so authors must manually create taxonomies and determine where classes belong in them.

We are exploring ways to combine the flexible, expressive reasoning of SE with the subsumption and classification algorithms of OWL-DL. Our approach is to define SE node types, functions, formulas, types, and regexes as OWL-DL classes. SE nodes would be instances of node type classes and SE links would be properties that relate those instances. We would use the SE reasoner to traverse the OWL KB much as we do now, but we would also be able to use OWL-DL's rich classification scheme.

9. CONCLUSION

We have described how enhanced semantic networks maintain the comprehensibility of a graphical knowledge representation while providing a flexible, unambiguous way to reason over that knowledge. Though not as powerful as generic rule systems, initial applications indicate that SE is sufficiently powerful to build conversational expert systems where interactivity and fast response times are paramount.

KB verification and validation are software engineering challenges that impact the manageability, reliability and, ultimately, the practicality of KBS's. We have developed static and dynamic analyses based on reasoning threads. Our approach is feasible and has good performance characteristics because of the simplified reasoning model that ESN's support. Our static analysis uses threads to verify and validate KB structure. Our dynamic analysis uses threads to guide the knowledge authoring process. Both analyses provide ways to characterize the effects of KB changes to knowledge engineers, which makes KB evolution easier and safer to perform.

10. ACKNOWLEDGMENTS

We thank previous and current contributors to the Semantic Engine project including, Karen Appleby, Leiguang Gong, and Moninder Singh.

11. REFERENCES

[1] Barker, V. and O'Connor, D. Expert System for Configuration at Digital: XCON and Beyond. *Communications of the ACM*, 32(3), March 1989.

[2] Borgida, A. On the relative expressiveness of description logics and predicate logics. Artificial Intelligence, 82 (1-2), pp 353-367, 1996.

[3] Brachman, R. and Levesque, H. The tractability of subsumption in frame-based description languages. *Proc. Of the 4th Nat. Conf. on Artificial Intelligence (AAAI'84)*, 1984.

[4] Chang, C., Combs, J. and Stachowitz, R. A Report on the Expert Systems Validation Associate (EVA). *Journal of Expert Systems with Applications*, 1(3), pp 217-230, 1990.

[5] DIMACS, Satisfiability Suggested Format, May 8, 1993. *Center for Discrete Mathematics and Theoretical Computer Science*, Rutgers University, Piscataway, New Jersey.

[6] Forgy, C. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Matching Problem. *Artificial Intelligence*, 19, pp 17-37, 1982.

[7] Ginsberg, A. Knowledge-Base Reduction: A new approach to checking knowledge bases for inconsistency and redundancy. *Proc. of 7th National Conference on Artificial Intelligence (AAAI'88)*, vol. 2, pp 585-589.

[8] Giunchiglia, E. and Sebastiani, R. Applying the Davis-Putnam procedure to non-clausal formulas. *Lecture Notes in Computer Science*, vol. 1792, Springer-Verlag, 2000.

[9] Minsky, M. A Framework for Representing Knowledge. *The Psychology of Human Vision*, P. Winston (Ed.), pp. 211-277, McGraw-Hill, 1975.

[10] Morgenstern, L., Mueller, E., Riecken, D., Singh, M. and Gong, L. Enhanced Semantic Networks: Hybrid Knowledge Structures for Reasoning. *IBM Research Report*, RC23436, Nov 16, 2004.

[11] Morgenstern, L. and Riecken, D. SNAP: An action-based ontology for e-commerce reasoning, *Proceedings, Formal Ontologies Meet Industry*, Verona, Italy, 2005.

[12] Nardi, D. and Brachman, R. An Introduction to Description Logics. *The Description Logic Handbook*, edited by Baader, F., Calvanese, D., McGuinness, D., Nardi, D. and Patel-Schneider, P. Cambridge University Press, 2003.

[13] O'Keefe, R. and O'Leary, D. Expert system verification and validation: a survey and tutorial. Artificial Intelligence Review, 7, pp 3-42, 1993.

[14] Onoma, A., Tsai, W., Poonawala, M. and Suganuma, H. Regression testing in an industrial environment. *Communications of the ACM*, 41(5), May 1998.

[15] Parnas, D. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12), pp 1053-1058, December 1972.

[16] Preece, A. Validation of Knowledge-Based Systems: The State-of-the-Art in North America. *The Journal for the Integrated Study of Artificial Intelligence Cognitive Science and Applied Epistemology*, 11(4), 1994.

[17] Preece, A. and Shinghal, R. Foundation and Application of Knowledge Base Verification. *International Journal of Intelligent Systems*, 9(8), pp 683-701, 1994.

[18] SAT4J, http://www.sat4j.org.

[19] Smith, M., Welty, C. and McGuinness, D. OWL Web Ontology Language Guide. *W3C Recommendation*, Feb 10, 2004. (http://www.w3.org/TR/2004/REC-owl-guide-20040210/)

[20] Suwa, M., Scott, A. and Shortliffe, E. An approach to verifying completeness and consistency in a rule-based expert system. *Technical report CS-TR-82-922*, Stanford University, 1982.

[21] Trowbridge, R. and Weingarten, S. Clinical Decision Support Systems. *Making Health Care Safer: A Critical Analysis of Patient Safety Practices*. Agency for Healthcare Research and Quality, Pub. No. 01-E058, July 2001.

[22] Tsai, W., Vishnuvajjala, R. and Zhang, D. Verification and Validation of Knowledge-Based Systems. *IEEE Transactions on Knowledge and Data Engineering*, 11(1), January 1999.

[23] Weusten, M. Validation: the key concept in maintenance of Legal KBS. *Proc. Of the 4th Internat. Conf. on Artificial Intelligence and Law (ICAIL'93)*. Amsterdam, 1993.

[24] Woods, W. What's in a link: Foundations for semantic networks. *Representation and Understanding: Studies in Cognitive Science*, edited by Bobrow, D. and Collins, A., Academic Press, 1975.

[25] Zlatareva, N. An Integrated Approach to Quality Assurance of Expert System Knowledge Bases. *Proc. of 2nd Internat. Conf. on Information and Knowledge Management*. Washington, D.C., 1993.