Agent-Mediated Knowledge Engineering Collaboration

Adam Pease, John Li
Teknowledge
[apease | jli]@teknowledge.com

Abstract

Abstract: Knowledge Management is most necessary and valuable in a collaborative and distributed environment. A problem with commercial knowledge management tools is that they do not understand at a deep level the content that they are managing. In this paper we discuss the System for Collaborative Open Ontology Production (SCOOP), which manipulates logic expressions and checks for redundancies or contradictions between the products developed by different engineers. SCOOP also includes an automated workflow process that supports recommendations for changes and voting to agree on changes.

Introduction

The current state of practice in knowledge management hinges on human review of content. Tools exist for organizing content or facilitating human collaboration but they do not understand the knowledge that they are managing. The System for Collaborative Open Ontology Production (SCOOP) works with logic expressions (among other knowledge products), which can be understood, to some degree, by a machine.

There are four primary features of the SCOOP collaboration process.

1. Knowledge is organized hierarchically as a set of files with inheritance of content. Consistency among these files is maintained *vertically* by enforcing that any assertion must not be contradictory with respect to files of knowledge that is inherited by the file where the assertion in question is to be placed. Consistency is encouraged *horizontally* by alerting individuals to contradictions or redundancies with respect to the files of their colleagues, and providing them a structured process for being aware of and resolving inconsistencies if they so choose.

2. Knowledge developers can influence their colleagues in several ways. They can register supervisory authority over another developer's content. They can register interest in particular keywords that are dynamically matched to their colleagues' content, which allows them to be alerted when new content matching the keywords is created. They can register interest in specific files or documents their colleagues are creating.

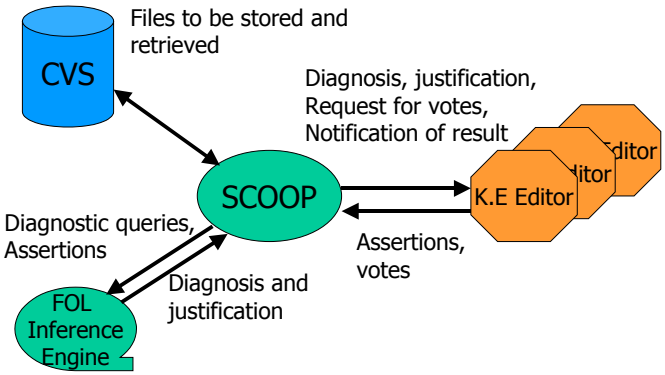
3. SCOOP assists developers in resolving conflicts and inconsistencies by identifying the statements by different developers that are problematic. The system accomplishes this by analyzing the results of theorem proving that detects the problem, and extracting only those statements that were authored directly, rather than automatically deduced. The full proof remains available for analysis by the authors if desired.

4. Authors have primary control over their own content but are given information that can help them maintain compatibility with the content created by their colleagues. When an author is alerted to a redundancy with respect to a colleague's content, he has several choices. He may withdraw the knowledge, elect to keep it, or vote to have it and the colleague's knowledge elevated to a more general document. A majority must vote to elevate the knowledge for this to occur. When a contradiction is discovered, the developer may either retract the knowledge, vote to keep it as knowledge which is contextually valid, although inconsistent with another view, or vote to have the colleague's knowledge removed. Such a vote is informational though since only the author, or someone who has registered authority over the author's content may change the author's content. This approach could of course be made stricter to be in keeping with a more authoritarian work environment.

General Architecture and Mechanism

A general architecture of SCOOP and its environment are shown in the figure below. To the knowledge authors, SCOOP is an invisible agent working at the background. Its existence becomes known to the authors only when conflicts or errors are detected in their products. To perform the error detection and limited correction functions described above, the SCOOP has an inference engine that it uses to process diagnostic queries. This architecture allows any inference engine to be used for this purpose as long as it can answer logic queries and provide justifications. Our current implementation of SCOOP uses a first order logic (FOL) theorem-prover that accepts KIF (Genesereth, 1991), the language and grammar in which the assertions created by users are written. With small modifications, SCOOP can employ other inference engines that accept other languages the end users use.

SCOOP depends on a distributed scheme of inference. Each knowledge developer runs a local inference engine that handles vertical consistency with the developer's products, and those products he uses directly. A central inference engine handles the detection of conflicts between knowledge developers. Each assertion made by its author is first verified locally as vertically consistent and then sent to SCOOP for horizontal consistency check. As required by this co-authoring task, when the developers start to work on their own products, their inference engines, as well as SCOOP's central inference engine, have the same background knowledge base loaded.



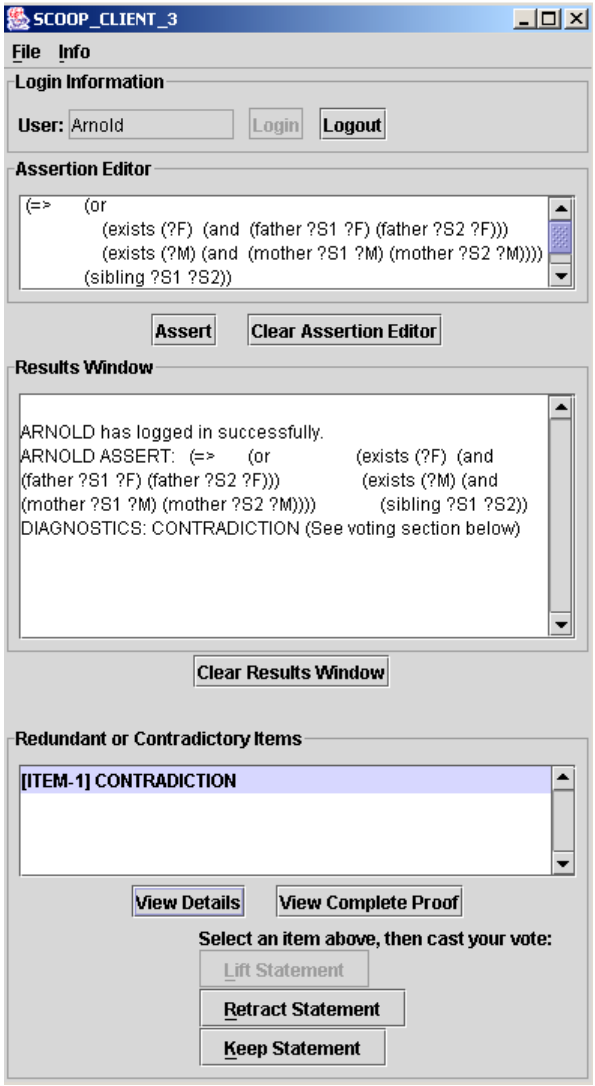
When a knowledge developer asserts a statement into a file, SCOOP is notified of this action. The content of the assertion is sent to SCOOP by the Knowledge Engineering (KE) Editor that the author uses to enter the assertion. SCOOP stores that assertion and its authorial information to its own temporary storage for possible future usage. When there are multiple assertions, SCOOP treats them in the order of their arrival. To detect redundancies, SCOOP asks a central inference engine whether it can prove that statement is true in the existing context. If it is not redundant, SCOOP will continue to ask whether the engine can prove the negation of the statement is true in the existing context. If both answers are negative, the statement will be asserted into the inference engine and becomes a part of the context to test the next statement by the same or different author. This mechanism of diagnosis sounds simple but is very powerful when more domain-specific axioms, diagnostic rules and term interpretations (such as synonyms and antonyms) are available in the background knowledge base. For example, SCOOP can detect a contradiction between (instance MyCar-1 FastMovingObject) and (instance MyCar-1 SlowMovingObject) when there is an axiom in the knowledge base on the oppositeness of the terms, FastMovingObject and SlowMovingObject.

If an error is detected, SCOOP will send the diagnosis and the justification to the author and other affected knowledge developers. The justification comes from the proof by SCOOP's inference engine but is not a full proof. It only lists the facts and axioms used by the inference engine to reach the diagnosis but not the further deductions. The full proof is generated by SCOOP as a web page that the user can view if desired. The usage of justification has two purposes. Besides giving a brief view of the assertions involved in the diagnosis, the justification also provides a base for an algorithm to identify the other authors who are related to the problem because of their contributions in the proof of the diagnostic result. Since the authorial information of each assertion is stored in SCOOP, SCOOP can easily identify the authors from the assertions and seek solution from only those who need to know, but not all authors.

An author has three choices with regarding to a redundancy warning: withdraw the knowledge, elect to keep it, or vote to have it and the colleague's knowledge elevated to a more general document. As to a contradiction case, the developer may either retract the knowledge, or to vote to keep it (in that case the contradictory knowledge will be removed by the colleague author). If the author does not want to retract the assertion diagnosed as redundant or contradictory to the existing knowledge base, a request to vote will be sent out by SCOOP to the need-to-know authors/reviewers, together with the result of the diagnosis message. When all votes are cast for a case, SCOOP will then tally the votes. The fate of the statement is determined based on the voting results and a pre-set policy. The voting authors will be notified of the decision made on the statement afterwards.

An Example of SCOOP Usage

The following example may provide a better understading of our current implementation of SCOOP. Suppose there are three knowledge engineers or subject matter experts, Joe, Jane and Arnold, co-authoring a knowledge base in a distributed environment. Each of them is working with a KE editor that is connected to its local inference engine. To work with SCOOP, a KE editor must be able to communicate with SCOOP and display messages as appropriate. We implemented a simple KE editor as shown in the figure below. Our KE editor has three major windows: Assertion Editor, Results Window and a window for Redundant and Contradictory Items. As the authors log in, the KE editor sends their user names to SCOOP and SCOOP establishes a user session for each of them. After an author adds assertions in the KE editor, the diagnostic result for the assertion is shown in the Results Window. If a redundancy or a contradiction is detected, a warning will appear and a new indexed entry will appear in the Redundant or Contradictory Items window. The author can see the detailed report (with justification) or the full proof by highlighting the item in the window and then making a choice on the View Details button or View Complete Proof button. The voting buttons are also enabled when a redundant or contradictory item is selected.



As an example, suppose that Joe defines that a `sibling` relationship is an irreflexive relation and his assertion gets OK from SCOOP:

```
(instance sibling IrreflexiveRelation)
```

In SCOOP's background, there is an axiom about the IrreflexiveRelation:

```
(=>
  (instance ?REL IrreflexiveRelation)
  (forall (?X)
    (not
      (holds ?REL ?X ?X))))
```

Jane is asserting some facts about the sibling relation:

```
(mother Bill Jane)
(mother Bob Jane)
(sibling Bob Bill)
```

SCOOP accepts each of these statements and returns an OK message.

Arnold tries to assert an axiom that defines the sibling relationship in terms of father and mother relationships:

```
(=>
  (or
    (exists (?F)
      (and
        (father ?S1 ?F)
        (father ?S2 ?F)))
    (exists (?M)
      (and
        (mother ?S1 ?M)
        (mother ?S2 ?M))))
  (sibling ?S1 ?S2))
```

As Arnold enters the assertion, SCOOP sends him the result of the diagnosis, which is a contradiction, and adds an entry in the windows below "[ITEM-1] CONTRADICTION". By highlighting this entry, Arnold can choose to view the detailed report or view the full proof. These two reports are as follows:

Sample XML-formatted Report

Note that some XML tags below have been removed for brevity.

```
<DETAILS:>
   <ITEM-ID:1>
   <assertion:
     (=>
       (or
         (exists (?F)
           (and
             (father ?S1 ?F)
             (father ?S2 ?F)))
         (exists (?M)
           (and
             (mother ?S1 ?M)
             (mother ?S2 ?M))))
       (sibling ?S1 ?S2))>
   <author: ARNOLD>
   <diagnostics: CONTRADICTION>
   <justification>
      <premises>
         <statement:(mother Bob Jane)>
         <statement:
           (=>
             (instance ?X0 IrreflexiveRelation)
             (forall (?X1)
               (not (holds ?X0 ?X1 ?X1))))>
         <statement:
           (instance sibling IrreflexiveRelation)>
      <conclusion>
         <statement:
           (not
             (=>
               (or
                 (exists (?F)
                   (and
                     (father Bob ?F)
                     (father Bob ?F)))
                 (exists (Jane)
                   (and
                     (mother Bob Jane)
                     (mother Bob Jane))))
               (sibling Bob Bob))>
```

Sample Proof

```
Result: There is 1 answer.
Answer 1:[definite] ?S2 = Bob, ?S1 = Bob
1. (mother Bob Jane)[KB]

2. (=>
     (instance ?X0 IrreflexiveRelation)
     (forall (?X1)
       (not (holds ?X0 ?X1 ?X1))))[KB]

3. (forall (?X0)
     (=>
       (instance ?X0 IrreflexiveRelation)
       (forall (?X1)
         (not  (holds ?X0 ?X1 ?X1))))) [2]

4. (forall (?X0)
     (or
       (not (instance ?X0 IrreflexiveRelation))
       (forall (?X1)
         (not (holds ?X0 ?X1 ?X1))))) [3]

5. (or
     (not (instance ?X0 IrreflexiveRelation))
     (not (holds ?X0 ?X1 ?X1))) [4]

6. (instance sibling IrreflexiveRelation)[KB]

7. (not
     (not
       (=>
         (or
           (exists (?X4)
             (and
               (father ?X3 ?X4)
               (father ?X2 ?X4)))
           (exists (?X5)
             (and
               (mother ?X3 ?X5)
               (mother ?X2 ?X5))))
       (sibling ?X3 ?X2)))) [Negated Query]

8. (forall (?X2 ?X1)
     (not
       (not
         (=>
           (or
             (exists (?X0)
               (and
                 (father ?X1 ?X0)
                 (father ?X2 ?X0)))
             (exists (?X3)
               (and
                 (mother ?X1 ?X3)
                 (mother ?X2 ?X3))))
         (sibling ?X1 ?X2))))) [7]

9. (forall (?X2 ?X1)
     (or
       (and
         (forall (?X0)
           (or
             (not (father ?X1 ?X0))
             (not (father ?X2 ?X0))))
         (forall (?X3)
           (or
             (not (mother ?X1 ?X3))
             (not (mother ?X2 ?X3)))))
     (sibling ?X1 ?X2 ))) [8]

10.(or
     (and
       (or
         (not (father ?X1 ?X0))
         (not (father ?X2 ?X0)))
       (or
         (not (mother ?X1 ?X3))
         (not (mother ?X2 ?X3))))
   (sibling ?X1 ?X2)) [9]

11.(or
     (sibling ?X1 ?X2)
     (not (mother ?X2 ?X3))
     (not (mother ?X1 ?X3))) [10]

12.(not (sibling ?X0 ?X0)) [5,6]

13.(or
     (sibling ?X0 Bob)
     (not (mother ?X0 Jane))) [11,1]

14.(and (= ?S2 Bob)(= ?S1 Bob)) [1,12,13]
```

Note that in the proof above, [N] after a formula means the formula is derived from the formula in proof step #N. [KB] means the assertion already exists in the knowledge base, either by assertion or in a preloaded background context.

The proof shows that the axiom developed by Arnold has a flaw that can lead to (sibling Bob Bob), violating the declared irreflexive nature of the predicate. The author may choose to retract it to resolve the conflict with the other axioms. If the author chooses to retain the statement, SCOOP will try to resolve the issue by sending a vote request to Jane and Joe because their products are involved in the proof.

Version Control and Change Notification

To perform the centralized knowledge harmonization functions described above, SCOOP is connected to a Concurrent Version System (CVS) repository that stores the files developed by all authors at different stages. The repository is organized into knowledge modules so the products at different development stages can be stored together under a theme. SCOOP allows users to specify relationships between themselves, others and knowledge products in the system that influences the workflow process employed when change or conflict occurs. A knowledge developer can designate another developer as the knowledge reviewer of his or her products. All users can also register their interests in the products of other authors via keywords. If there is any match of the keywords to the contents of the files in the repository, existing or in the future, the user will automatically get informed of the presence of the new or changed content. When an author submits a file for review, the reviewer will also automatically get a notice. The reviewers can vote to approve or reject a document and determine whether a file should pass the reviewing process. All the notifications and votes are carried via automatically generated email messages.

SUMO

SCOOP is especially useful when there is a large body of formal knowledge. The example in the previous section displays the use of a background axiom in the proof. One key to having informed and intelligent assistance to the knowledge developer is having existing knowledge that can support machine analysis of new content. We developed a large, formal ontology that can support that need.

The SUMO (Suggested Upper Merged Ontology) is an ontology that was created at Teknowledge Corporation with extensive input from the SUO mailing list, and it has been proposed as a starter document for the IEEE-sanctioned SUO Working Group. The SUMO was initially created by merging publicly available ontological content into a single, comprehensive, and cohesive structure (Niles & Pease, 2001) (Pease et al, 2002). As of January 2003, the ontology contains 1000 terms and 4000 assertions including 750 rules. The ontology can be browsed online (http://ontology.teknowledge.com), and source files for all of the versions of the ontology can be freely downloaded. SUMO has also been mapped, by hand, to the complete set of 100,000 WordNet (Miller et al, 1993) noun, verb, adjective and adverb word senses. This supports keyword matching within SCOOP.

Related Work

The Ontolingua (Farquhar et al, 1996) ontology server provides user and group access control that can facilitate group work. It also allows simultaneous access to ontologies and change highlighting. Some work detecting inconsistencies during ontology merging has been shown (Noy & Musen, 2000). Other relevant work includes (Elst, 2001) and (Dignum, 2002).

Previous work in collaborative ontology construction environment and recent developments in CSCW (Computer Supported Cooperative Work), knowledge sharing models, meaning negotiation approaches and ontological approach in knowledge management has provided a broad base and motivation for our research. However, SCOOP is unique in the following aspects:

(1) SCOOP concentrates on support for knowledge and ontology authoring and formation activities. It is not a general-purpose CSCW tool.

(2) SCOOP mainly supports collaborating authors who are formalizing knowledge in the same domains to create harmonious knowledge for the domains. It is not a general-purpose collaborative tool that brings the knowledge owners and the seekers together.

(3) SCOOP provides a conflict detection and limited conflict resolution mechanism on the contents the authors generate. It is not simply a distributed co-authoring environment that does not check the correctness of the contents.

(4) SCOOP not only provides notifications to the co-authors when conflicts are detected, but also conducts voting among the related authors and seeks resolutions.

Future Work

A primary area of effort is to make it possible for users to author content that SCOOP can check for consistency. We have developed a system for translating a restricted natural language to logic. Otherwise, we are faced with having developers author knowledge in formal logic, which would dramatically limit the applicability of this work.

A second area of effort is in improving the quality of proofs of contradiction and redundancy. We have done preliminary work in removing the application of the same axiom in a particular line of reasoning. Additional efforts are in removing proof steps that are conceptually very small, such as taking advantage of the associativity of the logical operator “and” to change the order of clauses. We can currently generate natural language from logic, as well as translating from language to logic, although considerable work remains in making this output more colloquial.

A potentially very difficult problem is in guessing how long to let a theorem prover run in order to find contradictions or redundancies. It is quite possible that however long the system tries to find a problem, that other undiscovered problems remain. Users are not going to be willing to wait indefinitely before their knowledge is declared consistent and compatible with that of their colleagues. As a result, SCOOP must consider that an undiscovered contradiction may at some point cause all subsequent proofs to declare the existence of an erroneous contradiction, since if the system is asked to prove a contradiction, and another real contradiction exists, the system will always respond “true”. This is a result of the fact that any fact holds from a contradictory knowledge base.

Future work will also include a more flexible and powerful method for specifying interest and relevance in knowledge products. While currently a simple keyword matching scheme is employed, the presence of formally encoded knowledge provides a rich basis for more powerful methods. We anticipate provide full logical queries to specify interest, and employing our theorem-prover to pose the query against new knowledge products.

Finally, we anticipate providing alternate encodings of knowledge to facilitate knowledge sharing. Our work on the DARPA DAML project (Pease et al, 2002) has allowed us to develop translators to convert formal ontologies to a form suitable for the semantic web. We have also developed a semantic web crawler and search system, which should facilitate knowledge discovery on a much larger scale than any individual collaboration system will allow.

References

Dignum, V., (2002), A Knowledge Sharing Model for Peer Collaboration in the Non-Life Insurance Domain, Proceedings of German Workshop on Experience Management, 7. 8 March 2002, Berlin, Germany. Published in Lecture Notes in Informatics of the German Society for Informatics. available at http://www.cs.uu.nl/people/virginia/publications.htm#y03

Elst, L., Abecker, A. "Domain Ontology Agents in Distributed Organizational Memories". In Working Notes of the IJCAI 2001 Workshop on Knowledge Management and Organizational Memories, August 2001, Seattle, Washington, USA, pp. 39-48.

Farquhar, A., Fikes, R., and Rice, J., (1996). The Ontolingua Server: a Tool for Collaborative Ontology Construction. Proceedings of Tenth Knowledge Acquisition for Knowledge-Based Systems Workshop. Available at http://ksi.cpsc.ucalgary.ca/KAW/KAW96/farquhar/farquhar.html#RTFToC15

Genesereth, M., (1991). ``Knowledge Interchange Format", In Proceedings of the Second International Conference on the Principles of Knowledge Representation and Reasoning, Allen, J., Fikes, R., Sandewall, E. (eds), Morgan Kaufman Publishers, pp 238-249.

Miller, G., Beckwith, R., Fellbaum, C., Gross, D., and Miller, K. "Introduction to WordNet: An On-line Lexical Database." 1993.

Niles, I & Pease A., (2001) "Towards A Standard Upper Ontology." In Proceedings of Formal Ontology in Information Systems (FOIS 2001), October 17-19, Ogunquit, Maine, USA, pp 2-9. See also http://ontology.teknowledge.com

Noy, N., and Musen, M., (2000). PROMPT: Algorithm and Tool for Automated Ontology Merging and Alignment. Seventeenth National Conference on Artificial Intelligence (AAAI-2000), Austin, TX.

Pease, A., Niles, I., Li, J., (2002), The Suggested Upper Merged Ontology: A Large Ontology for the Semantic Web and its Applications, in Working Notes of the AAAI-2002 Workshop on Ontologies and the Semantic Web.

Statement of Interest of the Authors

Adam Pease is Program Manager and Director of Knowledge Systems at Teknowledge Corporation. He led an integration team for the DARPA High Performance Knowledge Bases project and he participates in the DARPA DAML project. He was chair of the IJCAI-2001 Workshop on the IEEE Standard Upper Ontology and initiated Teknowledge's work on a Suggested Upper Merged Ontology proposal to the IEEE SUO group. He is the author of the Core Plan Representation (CPR), and the article "Knowledge Bases" in the Wiley Encyclopedia of Software Engineering. He worked previously at NASA/Ames and at the Naval Undersea Warfare Center. He holds M.S. and B.S. degrees in Computer Science from Worcester Polytechnic Institute. http://projects.teknowledge.com/apease

John Li is a senior scientist/project leader for Teknowledge's Knowledge Systems Division, currently working on DARPA's Rapid Knowledge Formation (RKF) project and DARPA Agent Markup Language (DAML) project. Previously, he worked for Teknowledge on DARPA's High Performance Knowledge Base (HPKB) project and other government projects. Dr. Li has his Ph.D. in Communication and Information Sciences from the University of Hawaii, 1992. Prior to joining Teknowledge, John was a Research System Developer, Research Corporation of the University of Hawaii.