An Agent-based Architecture for Software Tool Coordination*

Stephen Cranefield and Martin Purvis
Computer and Information Science,
University of Otago,
PO Box 56, Dunedin, New Zealand

29 July, 1996

Abstract

This paper presents a practical multi-agent architecture for assisting users to coordinate the use of both special and general purpose software tools for performing tasks in a given problem domain. The architecture is open and extensible being based on the techniques of agent-based software interoperability (ABSI), where each tool is encapsulated by a KQML-speaking agent. The work reported here adds additional facilities for the user to describe the problem domain, the tasks that are commonly performed in that domain and the ways in which various software tools are commonly used by the user. Together, these features provide the computer with a degree of autonomy in the user's problem domain in order to help the user achieve tasks through the coordinated use of disparate software tools.

This research focuses on the representational and planning capabilities required to extend the existing benefits of the ABSI architecture to include domain-level problem-solving skills. In particular, the paper proposes a number of standard ontologies that are required for this type of problem, and discusses a number of issues related to planning the coordinated use of agent-encapsulated tools.

1 Introduction

Every computer user must at some time have wished their machine was more "intelligent" and could reason about the task being performed by the user and the steps that must be performed to achieve that task. In particular, for many computer users, their day-to-day work involves the use of a variety of different software tools, developed independently and using different data representations and file formats. Coordinating the use of these tools, remembering the correct sequence of commands to apply each tool to the current

* An extended version of this paper will appear in the Proceedings of the Workshop on Theoretical and Practical Foundations of Intelligent Agents, Fourth Pacific Rim Conference on Artificial Intelligence, 1996 (to be published by Springer in the Lecture Notes in Artificial Intelligence series).

problem and incorporating new and modified tools into their work patterns adds additional demands on the user's time and memory. This paper discusses an agent-based architecture designed to remove this drudgery from the user. The paper also includes a discussion of related planning and ontological issues.

The architecture combines the techniques of agent-based software interoperability (ABSI) [1] with a planning agent and facilities for the user to describe the problem domain, the tasks that are commonly performed in that domain and the ways in which various software tools are commonly used by the user. Together, these features provide the computer with a degree of autonomy in the user's problem domain. The user can request the initiation of domain-level tasks and the system will plan and execute the appropriate sequence of tool invocations. If a task is not completed in one session, the system will remember the current state of the task and how any intermediate data files relate to the overall task. By simply requesting the continuation of the task, the user can cause the planner to begin from where it left off previously. Providing these abilities has implications for the nature of the planning agent and also suggests the necessity of developing ontologies for certain standard information storage formats.

This work places no requirements on the internal structure of agents in the system (which, apart from specialised planning and console agents, are simply encapsulations of existing software tools). Instead it focuses on the representational and planning capabilities required to extend the existing benefits of the ABSI architecture to include domain-level problem-solving skills.

2 Agent-based Software Integration

In today's heterogeneous software environments, with tools written at different times for various specific purposes, there is an increasing demand for interoperability among these tools [2]. Progress in this area has been made in the field of "Agent-Based Software Interoperability" (ABSI) [1], also known as "Agent-Based Software Engineering" [2]. This involves the encapsulation of software tools and information servers as "agents" that can receive and reply to requests for services and information using a declarative knowledge representation language KIF (Knowledge Interchange Format), a communication language KQML (Knowledge Query and Manipulation Language) and a library of formal ontologies defining the vocabulary of various domains.

The agents are connected in a "federation architecture" which includes special 'facilitator' agents such as the one developed by the Stanford University Computer Science Department's Logic Group [1]. Facilitator agents receive messages and forward them to the most appropriate agent depending on the content of the message (this is known as "content-based routing"). Agents are responsible for registering the type of message they can handle with the facilitator. Tools can be spread across different platforms, in which case there is one facilitator agent on each machine. These communicate with each other to forward messages from agents on one machine to those on another. The resulting agent system is open and extensible: a new tool can be added to the system by providing it with a 'transducer' program (that translates between KQML and the tool's own communications protocol) or a 'wrapper' layer of code (a KQML interface written in the tool's own

command or scripting language) and registering its capabilities with a facilitator agent.

3 Desktop Utility ABSI

ABSI projects discussed in the literature have to date largely focused on domains where the software tools to be integrated are complex and/or expensive to develop, *e.g.* concurrent engineering of a robotic system [3], civil engineering [4], electronic commerce [5], and distributed civic information systems [6]. In contrast with these large-scale integration efforts, the problem addressed in this paper is how to support interoperation among an evolving workbench or toolkit of general purpose utilities and special-purpose tools. Working with such a toolkit can typically be characterised as follows:

- A number of tasks are performed in sequence, possibly over an extended period of time and with significant time intervals between some of the tasks.
- Information recording the current state of the problem domain must be kept and updated as each task is performed.
- A variety of data formats and software tools are used to perform the different tasks. Relevant new tools may appear and existing tools may be replaced or upgraded over time.
- Some general-purpose tools may be used to support work in a number of different problem domains.

Typically, the problem domain is relatively simple and can be adequately defined using standard data modelling representations (*e.g.* the relational data model). Also, as some of the tools used may be general purpose utilities such as relational database management systems, programmable text editors and spreadsheets, for maximum reusability these should be encapsulated by agents that communicate in terms of the data formats the tools operate on. Thus an architecture to support this type of tool interoperation must include ontologies describing low-level data formats and how these can be related to higher-level representations of the domain. As each of the tools used may only perform a small part of the overall task, the overhead of coordinating the use of different tools into a coherent sequence represents a significant overhead. Thus planning support is needed to help the user select the appropriate sequence of tools to be used.

We will refer to this type of ABSI problem as "Desktop utility ABSI" to distinguish it from the large scale interoperability projects referenced above.

4 The Architecture

Figure 1 shows our architecture for Desktop Utility ABSI [7]. This is based on the federation architecture with the addition of two specialised agents: a planning agent and a console agent.

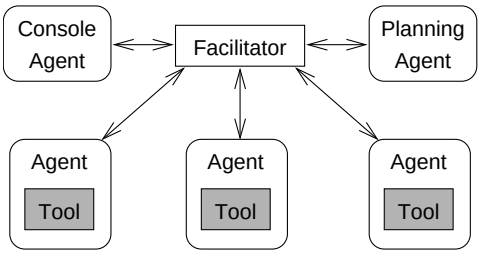

Figure 1: The desktop utility agent system architecture

4.1 The Planning Agent

For interoperating systems involving a few complex agents (large scale ABSI) the patterns of interaction between the agents are likely to be known in advance and limited in scope. In contrast, for desktop utility agent systems, there may be many simple agents each encapsulating a number of operations performed by various general-purpose tools. To achieve a user's high-level goal in the problem domain, it may be necessary for a number of different agents to perform actions in an appropriate sequence.

It would be tedious if the user were required to specify this sequence of agent invocations. However, deducing an appropriate sequence of agent actions to achieve a goal is an AI planning problem and the Stanford facilitator [1] does not support planning well as its knowledge is expressed using KIF which has no notion of state and time. Rather than attempting to extend the facilitator to perform planning, our architecture contains a specialised planning agent. Each agent is required to send the planning agent specifications of the actions it can perform (consisting of the action's preconditions and effects).

Figure 2 shows a specification[1] for an action performed by a database agent as part of a university course administration process (the predicates appearing in the specification are explained in Section 5). This describes an ability of the database agent to read in a specially formatted marks file (as produced by a utility used to systematically mark student's electronically submitted programming assignments) and to update the student marks database. Note that an explicit state variable appears in some of the pre- and post-conditions. The reason for this is explained in Section 5.1.

This agent action is designed for a specialised problem domain: its specification contains terms from the problem domain ontology. Agents may also encapsulate the general-purpose data manipulations of desktop utilities. For example, an agent might encapsulate the ability of the Excel spreadsheet to take as input a text file containing a column of numbers and produce a printed histogram of these numbers. In general, one agent wrapper must be provided for each application, and this will register with the planning agent all the functionalities of the underlying application that have been encoded in the agent. It is not envisaged that the full functionality of general packages such as spreadsheet or word processing applications could be expressed declaratively in a single specification. However, it should be possible to provide specifications for various simple computations that the application can perform.

[1] In this case no preconditions are retracted by the action, although in general this is possible.

Agent: `dbase-agent`
Action: `(add-marks ?f ?ass ?db)`
Description:
Add marks for assignment `?ass` in file `?f` to the database `?db`.

Preconditions:
```
(database-matches-datamodel ?db info202-dm)
(file-format ?f (delim #\, (listof string string number)))
(file-represents-relation ?f
   (select ASSESS (= cmptid ?ass))
   info202-dm
   (stuid cmptid mark)
   ?state)
```

Postconditions:
```
(database-includes-relation ?db
   (select ASSESS (= cmptid ?ass))
   info202-dm
   (do (add-marks ?f ?ass ?db) ?state))
```

Figure 2: Specification for a database agent action

Once an appropriate sequence of agent actions has been planned, the planning agent is also responsible for invoking agents to perform these actions for the user and to keep track of the current state of the system (*i.e.* what information is currently recorded and what it represents).

4.2 The Console Agent

In the desktop utility agent architecture the user interacts with a console agent that accepts requests expressed using the ontology of the user's various problem domains. In particular, the user can issue a request `(perform ?task)` where *task* is the (possibly parameterised) name of a problem-domain level task that the user wishes to be performed. This request is packaged up as a KQML message and sent to the facilitator (where it will be forwarded to the planning agent as described below).

4.3 Using the Architecture

The initial step in using the architecture involves defining a problem domain (which need only be done once for each domain). The user must define a relational data model defining the entities of interest in the domain. This is done by asserting facts (using the console agent) defining the relations in the domain. It is also necessary to assert facts describing the initial state of the system, *i.e.* what information is recorded in data files or databases

and how it relates to the domain-level data model.

The various domain-related tasks to be performed are specified by declaring their names and parameters and their precondition and goal states (in terms of the information recorded before and after the task is performed). The user may also provide a specification of possible expansions of each task into an ordered set of subtasks (see Section 6). In addition it is assumed that the user has previously developed or acquired agent wrappers for the various general or special-purpose tools used to support work in this domain. These wrappers must be equipped with planning operator style specifications of (a subset of) the possible actions these tools can perform (describing the actions' preconditions and effects in terms of the domain ontology or standard data storage ontologies as described in Section 5).

When the user requests a task to be performed via the console agent, the planner generates a sequence of agent actions that can satisfy the task specification. These actions are then invoked by the planner Note that while these actions are considered to be atomic by the planner they may involve the invocation of an agent-encapsulated tool which may possibly require a series of user interactions. During execution the planner keeps track of the current state of the system (in terms of the facts asserted and retracted by the actions' planning operators).

5 Required Ontologies

5.1 File Formats

In large scale ABSI projects, where there are a fixed number of interoperating tools that are specialised for a particular domain, it makes good sense to encapsulate each tool within an agent that speaks a high-level domain-related ontology. In contrast, in desktop utility ABSI there may be agents controlling various general-purpose tools that act at a relatively low data representation level (*e.g.* by performing manipulations on files). As these tools could be used in various problem domains it would limit their reusability in an ABSI framework if their agent wrappers could only declare their abilities in terms of a single high-level domain. Although for a general purpose tool it would be possible to generate separate agent wrappers corresponding to different domain ontologies, this would clearly be a duplication of effort. Instead, a generic tool should be described at the level at which it operates. Thus a utility that can manipulate text files should be described in terms of an ontology of text files.

There are a number of different levels at which the contents of text files are typically viewed, *e.g.* as a sequence of characters, a sequence of lines or a sequence of records. Figure 3 shows a hierarchy of ontologies being constructed for representing the contents of a text file in a declarative fashion, along with the type of facts used to describe the file contents at each level.

To relate the different viewpoints the ontologies also need to describe how to translate information between the different representational levels. For instance, the information that the file named "students.dat" is a text file with three string-valued fields delimited by commas might be represented by the following fact:

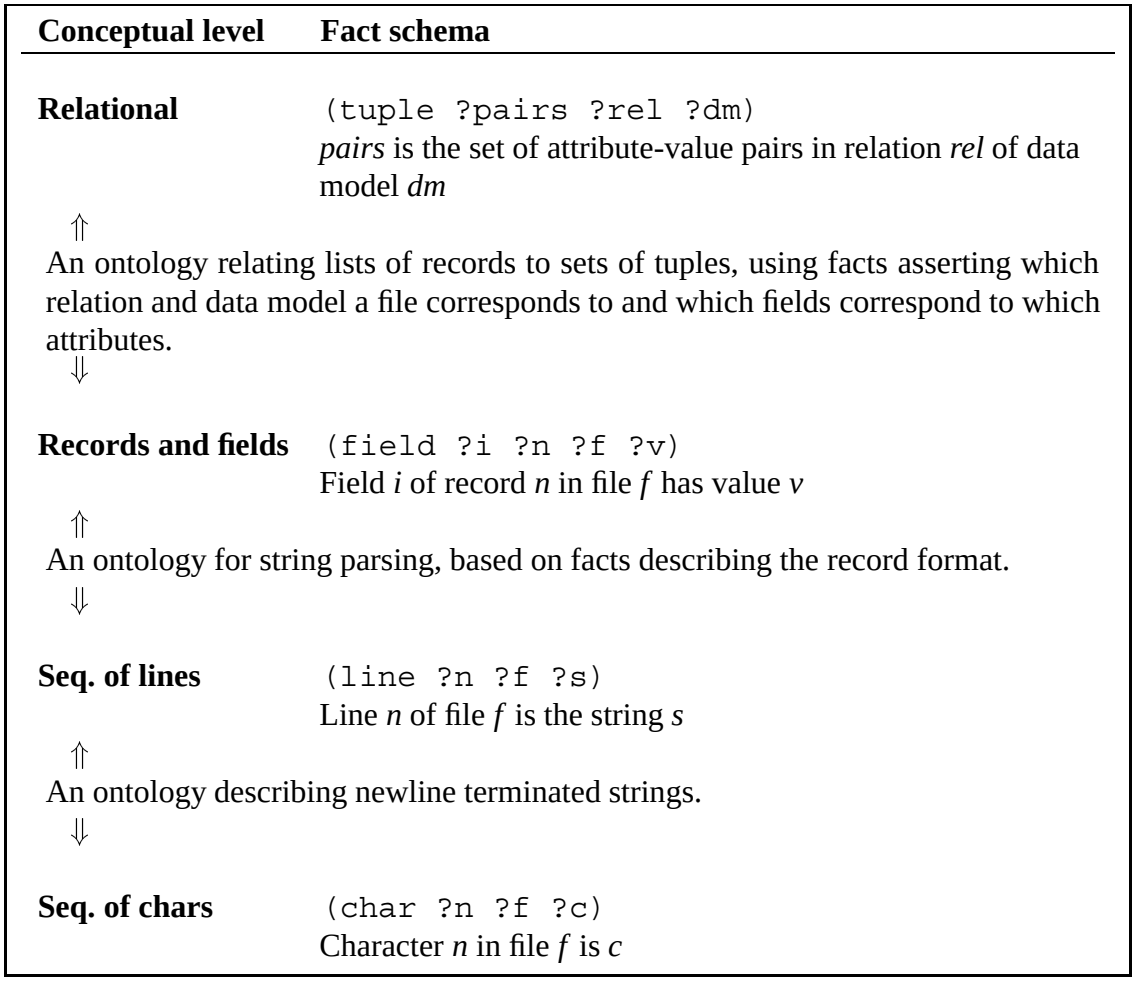

Figure 3: A hierarchy of text file ontologies and how they relate facts at different levels of abstraction

```
(file-format "students.dat"
   (delim #\, (list-of string string string)))
```

This predicate would be defined in an ontology by a formula stating how facts about records and fields can be inferred from facts about lines in files. For instance, the fact

```
(line 10 "students.dat" "9601234,Joe Smith,A0100001")
```

would allow the following three facts to be deduced:

```
(field 1 10 "students.dat" "9601234")
(field 2 10 "students.dat" "Joe Smith")
(field 3 10 "students.dat" "A0100001")
```

Thus the `file-format` fact above provides the information needed to translate between the line-based view of a file and the record-based view. Similarly, facts of the following form could be used to link the record-based and the relational views of a file:

```
(file-represents-relation
   ?file ?rel ?data-model ?att-list ?state)
```

As tuples in relations are *sets* of attribute-value pairs, whereas the fields in a record are ordered, the argument `?att-list` is required to specify the order in which the attributes appear in the file — this is a list which is some permutation of the attributes of the relation `?rel`.

The explicit mention of a state in this fact is necessary because in the type of tool use supported by this architecture, domain-level tasks may not be completed in a single session and there may be information stored in temporary files that are more up-to-date than other information sources. For example, in the university course administration domain, the user may run a marking utility to mark student assignments and then go home for the day before updating the course database with the marks recorded in the file that was output by the marking utility. At this point the database and the text file record information relating to different stages of the high level "mark assignment" task: the database does not reflect the fact that the latest assignment has been marked while the text file does. Not all facts need to have a state explicitly mentioned: only those recording that information is stored in a particular repository — it is necessary to specify which (possibly earlier) state of the system the information represents[2].

The ontologies described above are only *specifications* of standard terminologies that may be used by agents. While some agents may perform inference using the formulae in an ontology, all that is *required* of an agent is that its use of the terminology in an ontology be consistent with the ontology's logical theory. Defining ontologies describing different conceptual views of files, and how to translate between these levels of abstraction, does not imply that all file processing will be performed using an inference system. In practice, these ontologies — particularly at the lower levels — might be implemented by 'procedural attachment', so that telling a file manager agent a `file-format` fact such as the one above would result in that agent reading and parsing the file and sending a stream of `field` facts to all interested agents.

5.2 Data Models and Databases

In any ABSI project there needs to be at least one high-level ontology describing the problem domain (in some cases there may be several due to different agents having different views of the domain). In the example domain described above, the problem area involves student details, assignments and marks. An ontology for this domain can be created in various ways, *e.g.* an object-oriented model could be based on the Stanford ontology library's frame ontology [8]. However, like all applications involving a relational database, a domain model already exists: the relational data model developed for the database application. This could be easily described as an ontology if a standard ontology of relational data models were available. Such an ontology would define the concepts of base relations and their attributes and attribute domains, as well as a way of naming relations derived from the base relations. The discussion in this paper assumes

[2]The need to record state information in these facts was not realised when an earlier paper describing this architecture was written [7].

the existence of such an ontology (which is currently under development) with relations described using the relational algebra, *i.e.* relations can be built from the base relations using operations such as select, join and project. Users will define the relational data models for their problem domains by asserting facts naming the model and describing its base relations. The relational data model ontology will also allow the user to declare semantic information such as candidate keys, foreign keys, foreign key rules and default attribute values, thus supporting agents that can update derived relations ("view updates").

For the course administration example, using a domain ontology based on a relational data model, a tuple in a base relation STUDENT (recording student ID numbers, names and Novell NetWare IDs) could be represented by the following KIF fact:

```
(tuple (setof (stuid "9501234")
              (stuname "Joe Smith")
              (nw_id "A0100001"))
   student info202-dm)
```

where `student` names the relation and `info202-dm` names the particular relational data model (there could be more than one problem domain, and hence more than one data model in use at a time).

The relational data model ontology discussed above describes a conceptual model of a domain, whereas the text file ontologies discussed in the previous section describe the physical representations of data. Just as the text file ontologies refer to files, we need an ontology in which there is a concept of a database (as opposed to the conceptual model implemented by it). A database is a separate concept from a relational data model: a database could represent information from several relational models; conversely the information pertaining to a single data model could be split (or duplicated) across several databases. Therefore it is necessary to have an ontology for actual databases, defining (amongst others) the predicate:

```
(database-matches-datamodel ?database ?data-model)
```

This declares that the tables in the database match the relational data model specified (including any integrity checks implied by the foreign key information in the data model). This predicate can be implemented by procedural attachment to the database query language.

6 Planning Issues

The nature of desktop agent interoperation imposes a number of demands on the planner. The aim of this architecture is to remove as much manual coordination of tool use as possible from the user. For this multi-agent system to be as autonomous as possible, facilities need to be provided to assist the user in providing as much information about the problem domain as possible. In particular, the user may have common patterns of tool use that could improve the performance of the planner if it could make use of them. Such a facility is provided by hierarchical task-network (HTN) planners such as UMCP [9]. With this type of planner, the user can provide the planner with declarations

describing possible decompositions of tasks into ordered networks of subtasks. The basic planning step involves expanding a task into one of its possible networks of subtasks and then attempting to resolve any unwanted interactions between steps in the plan that were introduced by this expansion. A task expansion can also include subtasks of the form $achieve(g)$ where g is a goal. By providing methods for expanding 'achieve' tasks, the user can also provide the planner with (domain-specific) goal-directed planning capabilities.

The requirement that some ontologies include facts with explicit state parameters (discussed in Section 5.1) has implications for the planner. In particular, actions such as that shown in Figure 2 must have specifications with a state variable (s say) in the preconditions and a new state expression in the postconditions. In this paper an expression $(\texttt{do}\ \ a\ \ s)$ is used to represent the state resulting from performing action a in state s. In an HTN planning framework this expression is not necessarily a canonical expression for a state in the performance of the task being planned. The various tasks and subtasks defined for a problem domain must be equipped with specifications of their pre- and post-conditions and these may include state expressions of this form, except rather than atomic actions appearing as the first argument there will be the names of tasks or subtasks. This means that the same point in the performance of a task t can be represented by the state expression $(\texttt{do}\ \ t\ \ s_0)$ as well as many other more complex expressions containing the names of subtasks and/or atomic actions. The planner must be able to reason about this equivalence, and possible ways of doing this are currently being investigated.

An important characteristic of desktop utility ABSI is the mix of domain-specific and general-purpose tools. To use the information output by general-purpose tools as input to domain-specific tools, it is necessary to store facts stating how these low-level representations relate to the problem domain's data model. For example, Figure 2 shows the specification for an action that can be performed by a database-encapsulating agent in the university course administration domain: to update a student record database from a specially formatted text file. The action specification contains some pre- and post-conditions that are inherently related to the problem domain (university course administration). Other conditions are expressed using the lower level ontologies of file formats and databases. The planner may need to switch back and forth between these different levels of description as the following example shows.

Before the `add-marks` action can be performed a marking utility must be run to allow the tutor to systematically mark each student's electronically submitted programming assignment and create a file containing the marks. When encapsulated by an agent, the postconditions for this "mark assignments" action are as follows:

```
(file ?f
  (delimited #\, (listof string string string number)))
(file-represents-relation ?f
  (join STUDENT (project (select ASSESS (= cmptid ?ass))
                         (stuid mark)))
  info202-dm
  (stuid stuname nw_id mark)
  (do (mark ?ass ?f) ?state))
```

The format of the output file, and the relation it represents differ from that required by the `add-marks` action. The planner must find a way to achieve the preconditions of the `add-marks` action starting from the postconditions shown above. This is a text file manipulation process that takes the file produced as a result of running the marking program and converts it into a format suitable for adding information to the ASSESS table in the database. This can be performed by two operations at the text file level: deleting two fields of the file and then adding a new field containing the assignment name in each row. Figure 4 shows how these two file-level operations can be used to achieve the preconditions of the `add-marks` action starting with the postconditions of the `mark` action (the state expressions are not given in full).

To infer that this sequence of file manipulations can be used as a link in a plan involving higher level concepts requires the planner to drop down a level in the hierarchy of ontologies, generate a subplan at that level and then produce a specification of that subplan at the higher conceptual level. This reformatting process could be defined as a HTN planner 'task'; however, it would also be useful if the planner could solve this problem without such presupplied help from the user. This is an issue requiring further investigation.

7 Related Work

As discussed earlier, this work builds upon the ABSI federation architecture [2, 8].

SRI's Open Agent Architecture [10] allows agent-encapsulated desktop tools to interoperate in a distributed heterogeneous environment. Agents communicate via a distributed blackboard. User interface considerations are a focus of this work, while ontological issues and planning for high-level tasks are not addressed.

The SIMS architecture [11] is specialised for agents encapsulating distributed information sources. The agents are relatively complex, containing planning and learning components, and use domain and information source models developed using specialised knowledge representation tools.

Softbots [12] are stand-alone complex agents that can invoke various tools on behalf of the user. However, there is no facility for users to extend or alter a softbot's functionality.

8 Conclusion

This paper has presented an agent-based architecture for a type of software interoperability problem (desktop utility ABSI) different from the large scale ABSI projects discussed in the literature. In particular the use of a planning agent to automate the selection of actions to jointly achieve domain tasks has been discussed. It has been suggested that hierarchical task-network planning techniques be used so the user can provide guidance on how different tools can be combined for particular tasks. Combining the use of special-purpose tools with general-purpose utilities means that at times during the performance of a task, information corresponding to different stages of the task may be stored in different formats and the implications of this on the planner has been discussed.

```
(file ?f
   (delimited #\, (list-of string string string number)))
(file-represents-relation ?f
   (join STUDENT (project (select ASSESS (= cmptid ?ass))
                          (stuid mark)))
   info202-dm
   (stuid stuname nw_id mark)
   state1)
```

⇓ Delete columns 2 and 3 from file `?f`

```
(file ?f (delimited #\, (list-of string number)))
(file-represents-relation ?f
   (project (select ASSESS (= cmptid ?ass))
            (stuid mark))
   info202-dm
   (stuid mark)
   state2)
```

⇓ Insert new column 2 with value of `?ass` in every row

```
(file ?f (delimited #\, (list-of string string number)))
(file-represents-relation ?f
   (select ASSESS (= cmptid ?ass))
   info202-dm
   (stuid cmptid mark)
   state3)
```

Figure 4: The file manipulation process

Standard ontologies for data formats are required to facilitate the use of general-purpose tools with this architecture and the design of database and text file format ontologies have been outlined.

Currently work is continuing on the implementation of this architecture and example software agents to encapsulate the tools of the course administration domain. Further work involves clarifying the planning requirements for this architecture and elaborating the ontologies discussed. Also, ontologies for other common tools such as spreadsheets will be required ([13] may provide a useful starting point).

Acknowledgements

This research is supported by an Otago Research Grant. Thanks to Aurora Diaz for her ongoing work in implementing this architecture.

References

[1] N. Singh. A Common Lisp API and facilitator for ABSI: version 2.0.3. Technical Report Logic-93-4, Logic Group, Computer Science Department, Stanford University, 1993.

[2] M. R. Genesereth and S. P. Ketchpel. Software agents. *Communications of the ACM*, 37(7):48–53, July 1994.

[3] M. R. Cutkosky, R. S. Engelmore, R. E. Fikes, M. R. Genesereth, and T. R. Gruber. PACT: An experiment in integrating engineering systems. *Computer*, 26(1):28–37, 1993.

[4] T. Khedro and M. Genesereth. The federation architecture for interoperable agent-based concurrent engineering systems. *International Journal on Concurrent Engineering, Research and Applications*, 2:125–131, 1994.

[5] W. Wong and A. Keller. Developing an Internet presence with online electronic catalogs. Stanford Center for Information Technology, http://www-db.stanford.edu/pub/keller/1994/cnet-online-cat.ps.

[6] T. Nishida and H. Takeda. Towards the knowledgeable community. In *Proceedings of the International Conference on the Building and Sharing of Very Large Scale Knowledge Bases*, pages 157–166, 1993. http://ai-www.aist-nara.ac.jp/doc/people/takeda/doc/ps/kbks.ps.

[7] S. J. S. Cranefield and M. K. Purvis. Agent-based integration of general-purpose tools. In *Proceedings of the Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management*, December 1995. http://www.cs.umbc.edu/~cikm/iia/proc.html.

[8] Stanford Knowledge Systems Laboratory. Ontology Server Web page. http://www-ksl-svc.stanford.edu:5915/.

[9] K. Erol, J. Hendler, and D. S. Nau. UMCP: A sound and complete procedure for hierarchical task-network planning. In K. Hammond, editor, *Proceedings of the 2nd International Conference on AI Planning Systems*, pages 249–254, 1994.

[10] P. R. Cohen, A. Cheyer, M. Wang, and S. C. Baeg. An open agent architecture. In *Proceedings of the Spring Symposium on Software Agents*, Technical Report SS-94-03. AAAI Press, 1994. ftp://ftp.ai.sri.com/pub/papers/cheyer-aaai94.ps.gz.

[11] C. A. Knoblock and J. L. Ambite. Agents for information gathering. In J. Bradshaw, editor, *Software Agents*. AAAI/MIT Press, 1996. forthcoming. Also http://www.isi.edu/sims/papers/95-agents-book.ps.

[12] O. Etzioni, N. Lesh, and R. Segal. Building softbots for UNIX. Unpublished technical report, 1992. ftp://june.cs.washington.edu/pub/etzioni/softbots/softbots-tr.ps.Z.

[13] S. S. Ali and S. Haller. Interpreting spread sheet data for human-agent interactions. In *Proceedings of the Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management*, December 1995. http://www.cs.umbc.edu/~cikm/iia/proc.html.