Automating the Interoperation of Information Processing Tools

Stephen Cranefield, Emanuela Moreale*, Bryce McKinlay and Martin Purvis
Department of Information Science
University of Otago, Dunedin, New Zealand
{scranefield, mpurvis}@infoscience.otago.ac.nz,
em5@leicester.ac.uk, bryce.mckinlay@stonebow.otago.ac.nz

Abstract

This paper describes an agent-based architecture designed to support the interoperation of distributed and disparate information processing tools and resources. This work is based on the premise that many computer users have a tool kit of familiar software tools and information resources that must be used in conjunction to perform the user's information processing tasks. The architecture extends previous work on agent-based software interoperability, being based on the technique of encapsulating tools inside agent wrappers that communicate using high-level declarative messages. This, together with the use of a matchmaking 'facilitator' agent, allows agents to be added to the system or replaced at any time. Common patterns of tool interoperation can be automated through the use of a planning agent and a user agent that executes plans on behalf of the user. To address the unique features of information processing tasks, a novel extension of hierarchical task-network (HTN) planning is presented and its integration into the architecture is discussed.

1. Introduction

The recent rapid growth of the Internet, coupled with the decreasing cost and increasing computing power of desktop computers, has brought about a dramatic increase in the number of information and computational resources potentially available to a single user. Consequently, a number of research areas have emerged (or developed a new lease of life) due to this phenomenon, in particular, information retrieval from distributed and heterogeneous resources and the interoperation of disparate and distributed software tools. A common theme in this work is the use of an 'agent' metaphor, whereby different parts of a distributed system are encapsulated as service-providing agents communicating in a common declarative language. This research effort has led to the design of agent-communication languages (ACLs) such as Knowledge Query and Manipulation Language (KQML) [1] and the Foundation for Intelligent Physical Agents (FIPA) [2] ACL [3], and the development of agent architectures to support extensible and open agent systems [1].

This paper describes an architecture inspired by these developments and extended to address the problem of automating information *processing* tasks performed using distributed and disparate software tools and resources. This work is based on the premise that many computer users have a 'tool kit' of familiar software tools and information resources that must be used in conjunction to perform the user's tasks. If the tool kit does not consist of a single suite of software, designed by a single vendor and residing on the same hardware and software platform, there can be a significant time and memory overhead in performing the correct sequence of tool invocations, user-interface commands, data transfers and file format transformations required to achieve each information processing task. It can also be time-consuming for users to adjust their work practices to incorporate new tools in the tool kit or replace tools with updated versions or competing products.

An example of such a problem domain in the authors' environment is that of administering a university course. At the authors' institution, course information processing and management tasks include extracting initial class lists from the central database, adding and deleting students from the class roll, 'publishing' assignments and making any required data sets available, configuring an electronic assignment solution system, marking student assignments on-line, changing marks when errors in marking are detected, producing statistical summaries of the class marks, making exercise solutions and student marks available on the network, and producing a final end-of-semester report of the class marks. Information may be created, deleted or modified at each stage of the process. Currently these tasks are performed using a tool kit approach: the course administra-

* Now at Department of Engineering, University of Leicester, UK.

tor uses a number of different tools to perform the tasks, some being general-purpose tools he is familiar with, and some being specially written for work in this problem domain. Furthermore, over several years many of these tools have changed: the database software used has changed, the home-grown on-line assignment marking support utility has evolved through three generations, and other software tools (such as the spreadsheet utility) have been upgraded to newer versions.

To support users in this type of problem domain, the authors have previously proposed a "desktop utility agent" architecture [4–6] that extends previous work on agent-based software interoperability [1] by the addition of a specialised planning agent and a user agent that controls the automation of common sequences of actions of behalf of the user. This paper describes a prototype of an extended version of this architecture and in particular discusses the representation of information processing actions and plans and the requirements this imposes on the architecture. The discussion is illustrated with a simple example from the university course administration domain: automating the (possibly repeated) invocation of an on-line assignment-marking support utility and its associated resource retrieval actions in order to generate a class set of marks for an assignment.

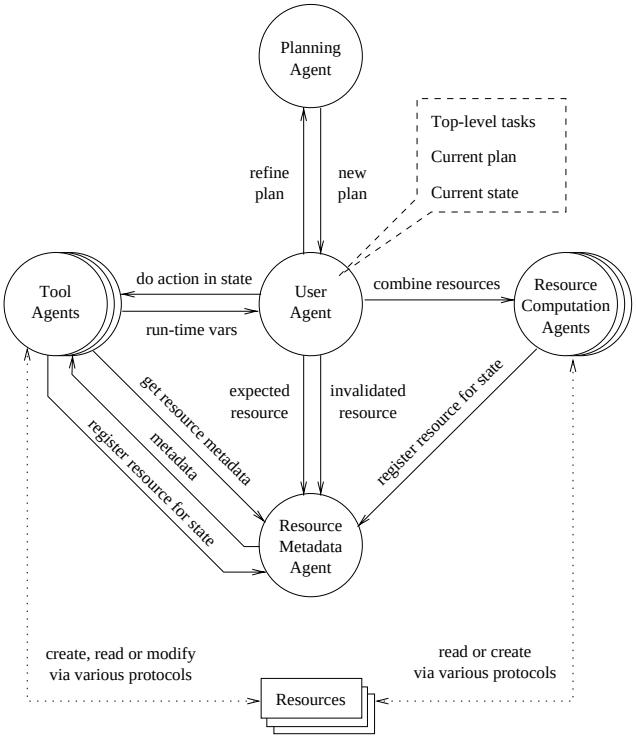


Figure 1. Agent interactions. All requests are sent via the facilitator as ''recruit-one'' KQML messages

2. Architecture overview

Figure 1 shows the agents in our architecture and their interactions. A *facilitator agent* (not shown) provides matchmaking services by routing requests for services to the appropriate agents (currently only the "recruit" mode of matchmaking [7] is implemented). The user interacts with the *user agent* to initiate a (possibly complex) task. This agent then asks a *planning agent* for a plan to be generated for that task and controls the execution of the resulting plan. The actions in the plan correspond to operations that can be performed by the agent-encapsulated tools (the *tool agents*). Available resources are registered with a *resource metadata agent* which records the intellectual content of the resource (in terms of a relational model of the problem domain), the data format used and the protocol required to access the resource. Because the information in a problem domain may change (e.g. the mark for a student's assignment), the resource metadata agent also 'time-stamps' each metadata record with the step (or steps) in the plan for which the resource is valid.

A plan may include 'resource computation links' that record how resources required for one action can be generated as some function of the resources produced by prior actions. For example, a resource may be needed for a relation that was produced in disjoint parts by two previous actions, thus a union operation is needed to combine them. For each type of resource computation that may appear in a plan there must be some *resource computation agent* that can perform this operation on resource files.

A prototype system has been built to test the architecture in the university course administration domain. Figure 2 shows the agents in this system (with the exception of a resource computation agent for text files, which is not shown). The *marking agent* is a wrapper around an existing tool that allows a tutor to systematically locate, compile, run and enter a mark for electronically submitted student programming assignments. The *relational data access agent* will be discussed later.

The system is implemented using Java and Amzi! Prolog [8]. A key feature is the development of our own *JATLiteBean*, a *Java Bean* [9] component that wraps around and extends the Java-based JATLite agent template from Stanford University's Center for Design Research [10]. In particular, the JATLiteBean provides

- an improved, easier-to-use interface to JATLite features, including receiving, parsing and sending KQML messages
- Simple configuration of agent capabilities and the au-

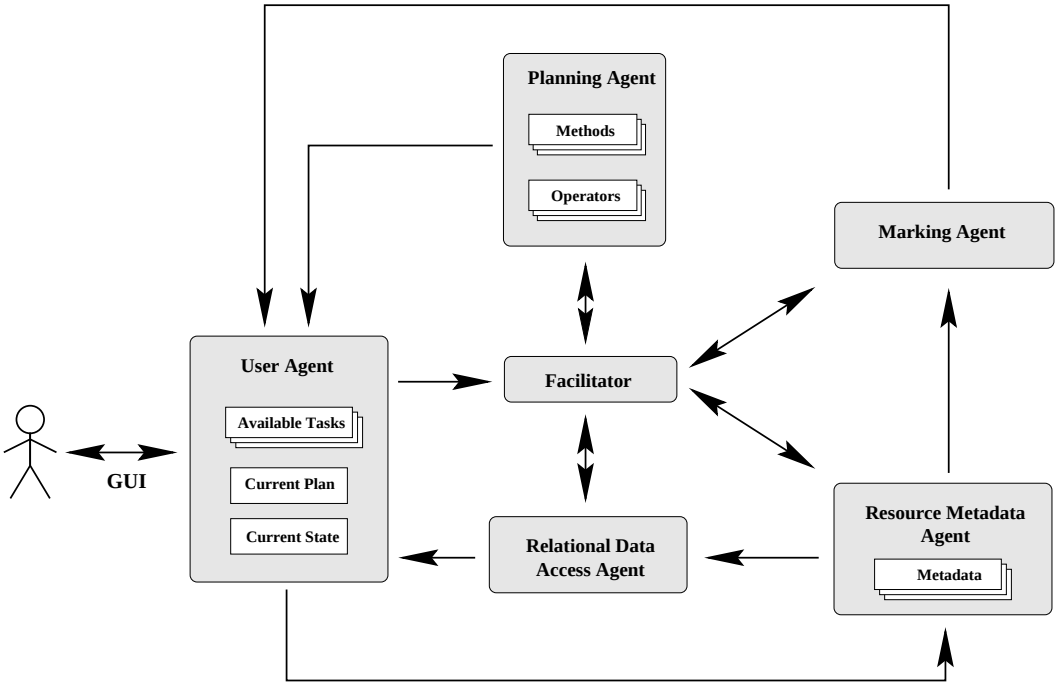


Figure 2. The prototype system architecture

tomated advertising of capabilities to the facilitator

- an extensible architecture for message handling

The JATLiteBean greatly simplifies the task of adding KQML functionality to a Java application or applet, to the point of simply dropping the component in GUI development environments. It also provides built-in domain-independent KQML functionality such as automatic handling of the `forward` performative and optional checking for the integrity of KQML messages (according to the 1993 KQML specification).

An important feature of the JATLiteBean is its extensible message-handling architecture. An agent can maintain a "thread of control" involving the transmission and receipt of many messages despite the asynchronous nature of KQML message-passing. This is achieved by allowing message handlers to be dynamically registered for expected incoming messages (such as replies to the agent's own requests). This is particularly important for the user agent which is responsible for executing plans — a process which involves requesting other agents to perform actions and then continuing its execution once a reply is received verifying that the action has been performed.

3. Automating information processing tasks

Although the current prototype system is being tested in a single domain, the architecture is designed to support the general problem of automating information processing tasks performed using distributed and disparate software tools and resources. This section outlines a number of such general issues that have guided the design.

3.1. A relational ontology

For multiagent systems to be open and extensible, they must have one or more commonly understood ontologies defining the meaningful expressions that may appear in messages. In general, a powerful knowledge representation language such as Knowledge Interchange Format (KIF) [11] may be required to define a domain. However, for many information processing problems the full representational power of first order logic is unlikely to be required. Furthermore, for many problems there is likely to be an existing formal description of the domain: the relational model used for the domain database(s). Therefore, this architecture assumes that the user has defined the domain ontology as a relational data model. This choice has influenced the model of action in which the planner is based and has allowed the frame problem to be addressed in a novel way (see Section 4).

3.2. HTN planning

From their experience with the manual interoperation of their software tool kit, users will already have a good understanding of the way in which their tasks can be broken

down into a sequence of actions to be performed by different tools, and the operations needed to transform data between different formats and machines. Therefore the planning support provided is not based on traditional goal-directed planning techniques, but instead is based on hierarchical task network (HTN) planning [12]. With this type of planning the planner is provided not only with operators describing the possible agent actions in the domain, but also with a set of parameterised abstract tasks that can be performed and a list of "methods", each describing a possible way to resolve a subtask into an ordered networks of subtasks. In this way a domain expert (the user in this case) can provide the planner with domain-specific knowledge.

3.3. Data format abstraction

To allow for tools and information resources to be changed and upgraded it is important that operators and plans abstract away from details about the data formats and access protocols of their resources. Therefore plans are only concerned with the intellectual content of resources (expressed as relational algebra expressions). At run time, the *resource metadata agent* stores metadata about the known information sources: their intellectual content, format, and their location and access protocol expressed as a uniform resource locator (URL). Agents needing input data must request resource information from the resource metadata agent, specifying the intellectual content and in addition the desired data format and access protocol. In this regard the agents are more fussy about their inputs than their associated planning operators: it is possible that the execution of a valid plan could fail because of a mismatch of output and input data formats at run time. The run-time system is responsible for dealing with this problem by providing an appropriate collection of data conversion agents.

If the resource metadata agent knows of a suitable resource it returns a URL to the requesting agent. If the resource is available only in a different format from that requested it should issue a request (via the facilitator) for a new copy of the resource to be generated in the desired format. However, at present this behaviour is not implemented.

With this separation of information content from format, new tools can replace old ones without affecting the rest of the system, provided that they can be given agent wrappers with the same interface as the tools they replaced.

3.4. Interleaving planning and execution

Execution of a task may involve invoking a tool that requires human interaction. The user's actions may affect the contents of any resources created by that tool. For example, in the marking task, the user enters students' marks on a form. The user can also have a direct effect on the future evolution of the overall task. For example, a user interacting with the marking tool is not required to mark every student's work in one session. Instead the tool can be repeatedly invoked until all submissions have been marked. To allow a route for information flow from the user's actions at execution time back to the planner, operators may include run-time variables [13] as parameters. These are instantiated when the corresponding agent action is performed. Planning and execution may be interleaved: the planner may generate a plan containing one or more unexpanded compound tasks. Once the initial actions in the plan have been executed, the instantiation of run-time variables may allow these tasks to be expanded further — this is done by the user agent passing the plan back to the planner to be refined.

One novel use of run-time variables in this architecture is to incorporate current information about the domain into the plan at run time. The *relational data access agent* can perform the action `eval_rel_exp(`*RelExp*`,` *RuntimeVar*`)` resulting in the instantiation of *RuntimeVar* to a term representing the set of tuples in the relation denoted by *RelExp*. This use of this feature is discussed later in the context of the example plan in Figure 6.

3.5. Recording information currency

Planning and executing information processing tasks requires explicit representation and reasoning about changing information resources. Information about the domain may be generated or altered as actions are performed (e.g. a student's mark may be corrected by a "change mark" action). Also, in a distributed system involving a number of different tools there may be duplication of information across various files and databases. For each known resource it is necessary to record how current it is. One solution would be to just 'forget' about resources that are known to be out of date. However, these can still be useful if used in the correct way, for example, an up-to-date file of student marks can be created by appending the latest output from the marking tool to the 'old' file of student marks. To ensure that this can be done correctly it is necessary to locate the correct version of the student marks resource. This capability is provided by the resource metadata agent: each metadata record contains a record of the range of world states for which the resource is valid. Requests for a resource URL should specify both the intellectual content (a relational algebra expression) and the state for which information is wanted.

The user agent is responsible for initiating the execution of each action in the plan. When it receives confirmation that the action has been performed it generates a new state name and notifies the resource agent that this new state has been reached. It also uses information in the plan to notify the resource metadata agent about any new or updated resources that will have been created by the agent that

performed the action. That agent will have created the resources and sent metadata describing them directly to the resource metadata agent. However, in case these messages are late arriving, the user agent must warn the resource metadata agent that they should be expected if they have not already arrived. The user agent also notifies the resource metadata agent of any resources that have become outdated as a result of the action (this information is represented explicitly in the plan).

3.6. Domain initialisation

When the architecture is used for the first time in a new domain, the user must provide the domain ontology, agent wrappers for the tool, operators describing the actions of these agents, and the domain tasks and methods. However, general purpose tools such as file format conversion utilities are useable across domains provided that their agent wrappers work in terms of lower level ontologies, e.g. an ontology describing file formats [4]. Providing user interface support for the definition of the domain ontology, operators and methods, and supporting the automatic generation of tool wrappers are important considerations, but at present we assume that the user and planning agents are already equipped with the appropriate domain knowledge and that the tools have already been encapsulated as agents.

4. The plan representation

This section will illustrate the design of the planning component of the architecture by showing an example planning operator and an example plan in the domain of university course administration. To clarify the structure of this domain, a simplified version of its relational model is shown in Figure 3. There are three base relations: `student`, recording details about students, `component`, describing the individual course assessment components (assignments, tests and the final exam), and `assess`, recording each student's mark for each component of the course.

Figure 4 shows an example operator specification. The operator `mark` represents the invocation of an interactive tool that allows a tutor to systematically run each student's submission for a particular assessment component (`Cmpt`) and record a mark for it. Not all submissions may be marked in one session, so the third argument is declared (by the use of the "!" prefix) to be a run-time variable. This will be instantiated at run time to be a term representing the set of student identification numbers for students whose work was marked during the execution of this operator.

The *affects* clause in the specification is used to solve the *frame problem* that must be addressed in any planning system [14]. This is the problem of describing and reasoning about actions so that both the effects of the action and the parts of the world state that are unaffected by the action can be efficiently determined. In this example, the *affects* clause states that the operator only alters the relation `assess`.

Table	`student`	
Attribute	`stu_id`	`stu_name`
Domain	String	String
Key	`stu_id`	

Table	`component`		
Attribute	`cmpt_id`	`out_of`	`weight`
Domain	String	Integer	Integer
Key	`cmpt_id`		

Table	`assess`		
Attribute	`stu_id`	`cmpt_id`	`mark`
Domain	String	String	Decimal(1)
Key	`stu_cmpt` = `stu_id`+`cmpt_id`		

Figure 3. The relational model for the course administration domain

The constraints appearing in operators describe how any relations affected by the action are altered. The new content of a relation may be only partially specified by an operator. For example, the marking action involves interaction with a tutor who decides the marks for the marked student assignments. Therefore, all that is known about this operator is that afterwards a resource exists that contains marks for all programs that were marked. The actual marks are not (and can not) be specified by the operator.

In the `mark` operator, the first two constraints specify how the operator changes the relation `assess` (shown in diagrammatic form in Figure 5). This operator is declared to create new information in the relation `assess`: marks for the assessment component `Cmpt` for all students in `MarkedIDs`. The first constraint declares that the contents of the `assess` relation after the operator executes is the union of the contents of `assess` beforehand and a set of new tuples represented by the variable `NewData`. The actual value of `NewData` is unknown until runtime, but the second constraint states that this relation will consist of a tuple for each student in `MarkedIDs`, with each tuple having its `cmpt_id` attribute equal to `Cmpt`. More precisely, the constraint states that the set of values in `NewData` for the key `stu_cmpt` — consisting of the pair of attributes `(stu_id, cmpt_id)` — is equal to the cross product of `MarkedIDs` and the singleton set `{Cmpt}`.

Note that operator constraints are not intended to be used for reasoning within the planner. However, they should be taken into account when designing task-expansion methods for the planner, which may include constraints that *are* evaluated during planning (see, for example, the bottom

mark(Cmpt, ToMarkIDs, !MarkedIDs)

Affects: assess.
Variables
NewData: (a relation with the same structure as assess)
Constraints
$\text{assess}' = \text{assess} \dot{\cup} \text{NewData}$,
$\text{key\_values}(\text{NewData}, \text{stu\_cmpt}) = \text{MarkedIDs} \times \{\text{Cmpt}\}$,
$\text{MarkedIDs} \subseteq \text{ToMarkIDs}$.
Input resources
R1 : ontology = course_data,
content =select(student, [stu_id $\in$ ToMarkIDs]).
Output resources
R2 : ontology = course_data,
content =select(assess, [stu_id $\in$ MarkedIDs, cmpt_id = Cmpt]).

Figure 4. The operator for the marking tool ('$\dot{\cup}$' represents disjoint union)

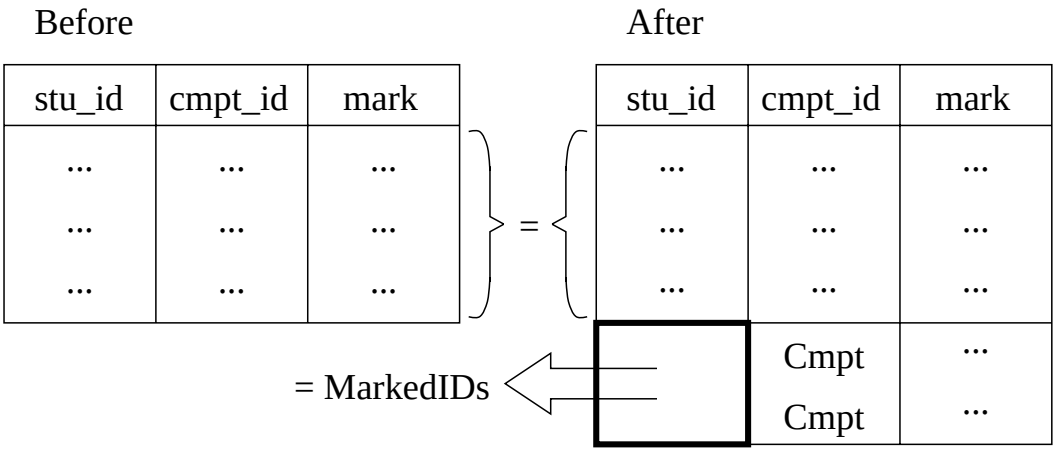


Figure 5. The effects of operator mark(Cmpt, ToMarkIDs, !MarkedIDs) on the relation assess

level of Figure 6). The operator constraints are also used to determine resource computation links between nodes in a method. At present this is done manually, but it should be possible to compute the method constraints and resource computation links automatically by reasoning about operator constraints.

An example plan is shown in Figure 6. The representation extends that used in standard HTN planning by the addition of constraints and the explicit representation of resources and the computations required to generate new resources from old. However, unlike standard HTN planning, our plans are currently restricted to be linearly ordered.

In the plan depicted the task `mark(ass1)` has been expanded into an action `eval_rel_exp(...)` and another task `mark(ass1, AllIDs)`. The designer of the task-expansion method for mark/1[1] has included a link between its two child nodes specifying that the required input resource to the mark/2 task (represented by the expression *exp1*) is in this situation equivalent to the complete student relation. This is because the run-time variable `AllIDs` will be instantiated by the relational data access agent (which executes the `eval_rel_exp` action) to a term containing all the student IDs currently in the student relation, and so *exp1* reduces to `student`.

A second method has then expanded the mark/2 subtask into a constraint, an action (corresponding to the marking tool operator shown above) and a recursive call to the mark/2 task which is currently unexpanded. This method includes a 'resource computation link' describing how the input resources for this recursive call can be computed from other resources that are known to exist (in the figure the arguments to the select operation have been omitted for brevity). When interpreting a plan, the user agent is responsible for requesting that these resource generation actions are performed.

Once the two actions in this plan have been executed, both run-time variables `AllIDs` and `MarkedIDs` will be instantiated, the constraint will then cause `RemainingIDs` to be instantiated and the user agent will return the plan to the planning agent for further expansion of the remaining mark/2 task. Through this combined use of run-time variables, a recursive task-expansion method and the interleaving of planning and execution, it is possible to generate iterative behaviour where the number of invocations of the marking tool necessary to complete the execution of this plan is solely determined by the user at run-time.

5. Further work

At present the planner for this architecture has not yet been implemented, as development has focused on the other aspects of the system while the design of the operator and plan representations were refined.

An important issue that must be addressed is to provide good user interface support to help a user set up a new domain for use with this architecture. This involves specifying the domain relational model, providing wrappers for the tools used and their corresponding operators, and defining the tasks and task-expansion methods to be used in planning. It is crucial that the benefits of automation are not outweighed by the initial time and cost of the domain initialisation phase.

A full implementation should include agents that can assist with the translation of data formats and the translation of messages from one ontology to another. For now, we assume that there is a single domain ontology and that all resources are delimited text files. If there were resources with various formats and access protocols, it would be necessary to consider the formats in which an action's input resources are available when selecting an agent to perform that action. Different agents might implement the same action for different resource data formats. This ability could be supported if the facilitator consulted the resource metadata agent before deciding where to forward a "perform action" message from the user agent.

Analysis of the type of automation support required in this architecture has focused on the university course administration domain. Other information processing domains should be investigated to see if they have additional requirements that are not currently catered for.

6. Related work

Various distributed information retrieval projects are discussed in references [15] and [16].

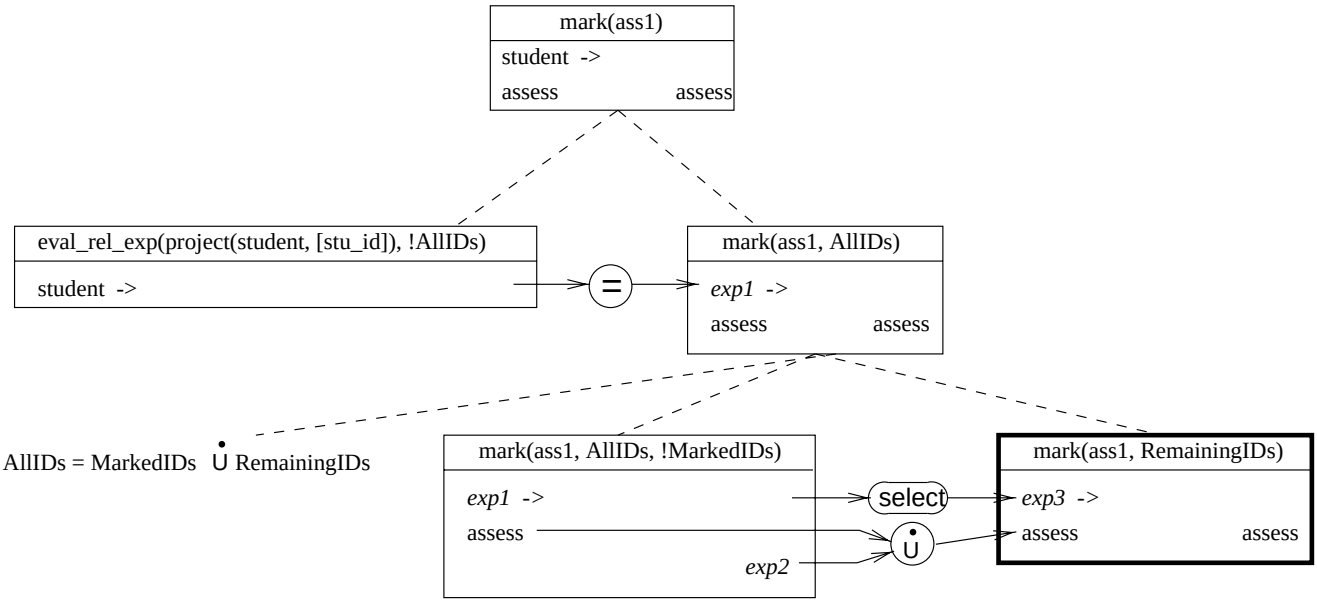
There are a number of planners designed to plan for gathering information from large dynamic networks of information sources ([17–20]), but none of these are designed with information processing tasks specifically in mind.

XII [17] is a general-purpose planner extended to describe actions that sense the world as well as causal actions. Although its actions can change the world, it makes a distinction between information goals and causal goals. It could probably be applied to information processing tasks but its action language is not designed to describe such domains succinctly.

Sage [18], the planner used in the SIMS project [21], is a general-purpose planner adapted to the problem of efficiently accessing multiple information sources in order to satisfy information gathering queries. It does not include a mechanism to model actions that change the information state.

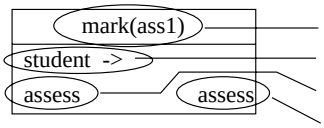
Occam [19] is an special-purpose algorithm designed for the same task as Sage. It models the available information

1. Tasks are identified using the Prolog notation *Name/Arity*.



exp1 = select(student, [stu_id in AllIDs])
exp2 = select(assess, [cmpt_id = ass1, stu_id in MarkedIDs])
exp3 = select(student, [stu_id in RemainingIDs])

KEY



Name of action or compound task
Precondition resource that is still valid after task execution
Precondition resource that is invalidated by task
Resource created by task (in this case, the old resource is no longer valid as the task has changed the information state)

!Var Runtime variable Var at point of instantiation

= Output resource is known to be equivalent to input resource

Op Output resource can be produced from input resources by operation Op (∪• represents disjoint uniom)

Figure 6. An example plan (the bold node is an unexpanded compound task)

as a relational database schema, but as Occam plans for information gathering from an unchanging world, this information model is regarded as static. The available information resources are modelled by associating information retrieval actions with the relations of the world model that are returned when these actions are executed.

Williamson *et al.* [20] extend the HTN planning paradigm by explicitly modelling a task's *provisions* (these are named interface 'slots' with an attached queue for storing incoming values), its *outcome* (indicating the result status of the task) and its *result* (a value produced by executing the task). Task networks are extended to include links between the results and provisions of tasks, indicating a flow of information. This mechanism is claimed to unify and generalise the methods by which operators can obtain information in traditional planning frameworks: by parameter binding, the passing of information from other operators via the world state, and through the use of run-time variables. Provisions also have a role to play in controlling the execution of plans, with primitive tasks being (re)activated whenever all their required inputs are available.

Our framework takes a different approach to representing and reasoning about information flow between actions. Resources are described in terms of their intellectual content described as a relational expression. Together with our use of the resource metadata agent this allows agents to use any available resource that contains the required information — the provider of the information does not need to be hard-wired into the plan.

We generalise the representation of links between different actions' output and input resources by allowing specified functions to transform and/or combine existing resources to produce new ones.

Finally, there is no aspect of plan execution intertwined with our representation of resources. As discussed in Section 4, iterative behaviour can be achieved through the use of run-time variables and recursive task-expansion methods.

7. Conclusion

An architecture has been developed to provide automated support to users in the general problem of performing information processing tasks where a variety of disparate software tools and resources must be used. This type of domain has different requirements from the types of applications addressed by existing work in agent-based software interoperability (in particular information retrieval from distributed sources and the interoperation of complex engineering design applications). In particular, this architecture must support a model of action that describes how resources change as domain actions are performed. It is necessary to keep track of the changing state of the system as actions are performed and the status of resources as they become updated or outdated. A novel extension of the HTN plan representation has been developed to support planning in this domain.

References

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

[2] Foundation for Intelligent Physical Agents homepage. http://drogo.cselt.stet.it/fipa/, 1998.

[3] FIPA 97 specification documents. http://drogo.cselt.stet.it/fipa/spec/fipa97/fipa97.htm, 1997.

[4] 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.

[5] S. J. S. Cranefield and M. K. Purvis. An agent-based architecture for software tool coordination. In L. Cavedon, A.S. Rao, and W. Wobcke, editors, *Intelligent Agent Systems: Theoretical and Practical Issues*, number 1209 in Lecture Notes in Artificial Intelligence, pages 44–58. Springer, 1997.

[6] A. C. Díaz, S.J.S. Cranefield, and M. K. Purvis. Planning and matchmaking in a multi-agent system for software integration. *Mathematical Modelling and Scientific Computing*, 8, 1997.

[7] D. Kuokka and L. Harada. Matchmaking for information agents. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, volume 1, pages 672–678, 1995.

[8] Amzi! Inc. WWW home page. http://www.amzi.com/.

[9] Java beans specification. http://java.sun.com/beans/docs/spec.html.

[10] JATLite Web pages, Center for Design Research, Stanford University. http://java.stanford.edu/, 1997.

[11] Knowledge Interchange Format specification. Working Draft, ANSI X3T2 Ad Hoc Group on KIF, March 1995. http://logic.stanford.edu/kif/specification.html.

[12] S. Kambhampati. A comparative analysis of partial order planning and task reduction planning. *SIGART Bulletin*, 6(1):16–25, 1995.

[13] J. Ambros-Ingerson and S. Steel. Integrating planning, execution and monitoring. In *Proceedings of the 7th National Conference on Artificial Intelligence (AAAI-88)*, pages 735–740, 1988.

[14] P. J. Hayes. The frame problem and related problems in artificial intelligence. In J. E. Allen, J. Hendler, and A. Tate, editors, *Readings in Planning*, pages 588–595. Morgan Kaufmann, 1990.

[15] G. Wiederhold, editor. *Intelligent Integration of Information*. Kluwer Academic Publishers, 1996. (a special double issue of the Journal of Intelligent Information Systems, vol. 6(2–3), June 1996).

[16] M. H. Huhns and M. P. Singh, editors. *Readings in Agents*. Morgan Kaufmann, 1998.

[17] K. Golden, O. Etzioni, and D. Weld. Omnipotence without omniscience: Efficient sensor management for planning. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, pages 1048–1054, 1994.

[18] C. A. Knoblock. Planning, executing, sensing, and replanning for information gathering. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, volume 2, pages 1686–1693, 1995.

[19] C. Kwok and D. Weld. Planning to gather information. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, 1996.

[20] M. Williamson, K. Decker, and K. Sycara. Unified information and control flow in hierarchical task networks. In *Proceedings of the AAAI-96 Workshop on Theories of Planning, Action, and Control*, 1996.

[21] C. A. Knoblock and J. L. Ambite. Agents for information gathering. In J. Bradshaw, editor, *Software Agents*. AAAI/MIT Press, 1997.