HTN Planning for Information Processing Tasks

Stephen Cranefield
Department of Information Science, University of Otago,
PO Box 56, Dunedin, New Zealand
Fax: 64 3 479 8311
scranefield@infoscience.otago.ac.nz

Abstract

This paper discusses the problem of integrated planning and execution for tasks that involve the consumption, production and alteration of relational information. Unlike information retrieval problems, the information processing domain requires explicit modelling of the changing information state of the domain and how the validity of resources changes as actions are performed. A solution to this problem is presented in the form of a specialised hierarchical task network planning model. A distinction is made between the information processing effects of an action (modelled in terms of constraints relating the domain information before and after the action) and the actions' preconditions and effects which are expressed in terms of required, produced and invalidated resources. The information flow between tasks is explicitly represented in methods and plans, including any required information-combining operations such as selection and union.

The paper presents the semantics of this model and discusses implementation issues arising from the extension of an existing HTN planner (SHOP) to support this model of planning.

Keywords: HTN planning, information processing, integrated planning and execution

HTN Planning for Information Processing Tasks

Stephen Cranefield
Department of Information Science, University of Otago,
PO Box 56, Dunedin, New Zealand

Abstract

This paper discusses the problem of integrated planning and execution for tasks that involve the consumption, production and alteration of relational information. Unlike information retrieval problems, the information processing domain requires explicit modelling of the changing information state of the domain and how the validity of resources changes as actions are performed. A solution to this problem is presented in the form of a specialised hierarchical task network planning model. A distinction is made between the information processing effects of an action (modelled in terms of constraints relating the domain information before and after the action) and the actions' preconditions and effects which are expressed in terms of required, produced and invalidated resources. The information flow between tasks is explicitly represented in methods and plans, including any required information-combining operations such as selection and union. The paper presents the semantics of this model and discusses implementation issues arising from the extension of an existing HTN planner (SHOP) to support this model of planning.

1 Introduction

This paper discusses the problem of integrated planning and execution for tasks that involve the consumption, production and alteration of relational information by a collection of disparate information processing tools and resources. Distributed object technologies such as CORBA [1] and agent-based software interoperability techniques [2] have made it easier to build information systems from distributed components, but with the additional complexity such distribution brings there is a need to support the user by automating much of the interactions between components.

Other researchers have investigated the use of planners in software agent frameworks [3–6], but this work has primarily focused on planning purely for information *gathering* actions (which do not change the world state except to increase the agent's knowledge of an external but static body of information). The work presented in this paper is concerned not only with planning for information gathering tasks, but also with planning for information *processing* tasks, where information can be created or altered by the actions of agents. This paper discusses the planning component of an architecture designed to address this issue [7] (see Fig. 1). This

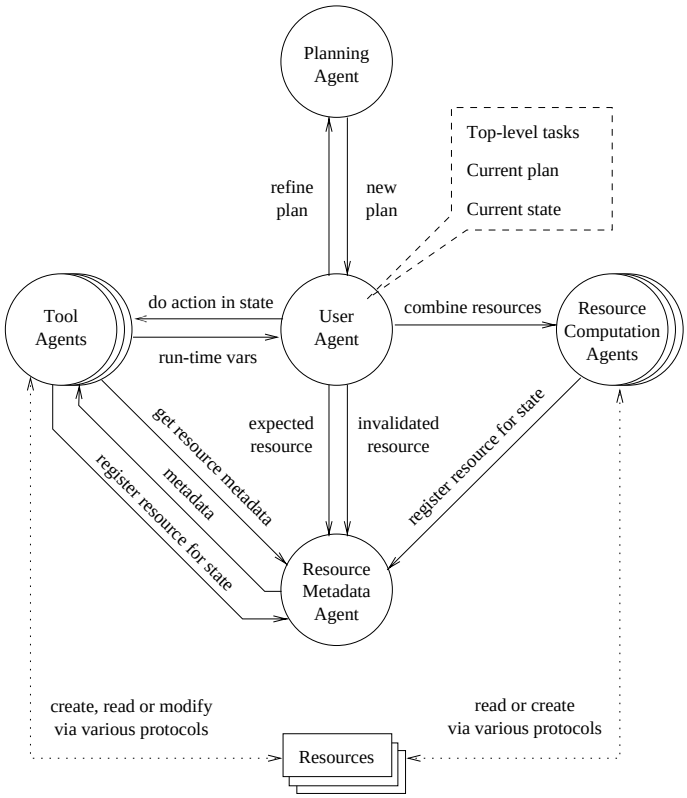


Figure 1: The agent architecture. All requests are sent via a facilitator agent (not shown).

architecture extends previous work on agent-based software interoperability [2] 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. In addition, a resource metadata agent records information about the available resources, the information they contain and the (possibly past) state of the system that this information corresponds to. Plans can involve using resource computation agents to combine existing resources using operations such as selection, union and join to produce new resources. The operators in the planner correspond to the invocation of agent-encapsulated software tools. The execution of these actions can feed information back into the plan through the instantiation of run-time variables [8]. Planning and execution is interleaved so that after the instantiation of a run-time variable the plan can be sent back to the planner to be expanded further.

2 Planning for Information Processing Tasks

2.1 A Relational Domain Information Model

In order to model information processing actions we must first choose a representation for the information structure of our problem domains. This architecture assumes that the user has de-

fined the domain information structure as a relational data model, consisting of a set of *base relations*. Operators may have resource pre- and postconditions described by these base relations or more complex relational algebra expressions. It is assumed that the relational data model is well-designed so that the base relations represent the most important clusters of information that are affected by actions. Operators declare which base relations in the domain they affect and this allows the frame problem to be addressed in a novel way (see Sect. 5).

2.2 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 information processing 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 this planning framework is based on hierarchical task network (HTN) planning [10]. 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.

2.3 Modelling Information Processing Tasks

The application of this research is the generation of plans to coordinate the use of information processing tools to effect an overall information processing goal. This leads to two observations: (1) the changes made to the domain's information state made by actions must be modelled by the corresponding operators; and (2) it is not appropriate to model these effects using the traditional planning representation of change via operator postconditions that assert and delete facts in the world model. This is the wrong level of abstraction. A planner can be regarded as simulating the real world execution of the plan being generated, at least in as much detail as is required to ensure that the desired goals will be achieved. For a plan involving physical actions, the final state cannot be confused with the real world goal state that is required — a memory structure representing the assertion $on(a, b)$ is rather different to the physical reality of one block stacked on top of another. It is necessary to execute the plan in order to achieve the physical goal.

In information processing problems this is not the case. If the planner exactly modelled the desired information processing to be performed, there would be nothing left to do once the plan was generated. Clearly, the role of the planner is not to process the information but to decide which tools can achieve the required information processing goals (for instance, to model a tool used to systematically mark student assignments submitted on-line it is enough for the operator to state in its postconditions that a mark now exists for every student; the actual marks need not be represented). To support this abstraction away from the details of information processing actions, the action representation proposed here models the "information state" of a domain in terms of a relational data model, and allows operators to include constraints that relate the contents of the information state before and after the action is executed. This abstract information state should not be confused with the information resources that may be required or produced by an action

(these are a separate part of an operator specification). It is possible (but pointless) for an action to change the information state (e.g. by giving a student a mark for an assignment), but neglect to record this information anywhere.

A more realistic example of this distinction between information state effects and resource pre- and postconditions occurs in the assignment marking example discussed in this paper. The tool for systematically accessing and invoking students' programming assignments requires an input resource corresponding to the relation *student*. This contains the details of all students in the class. Its output is a resource containing the new marks generated in the current marking session. The operator representing a session using this tool must model the fact that the *assess* relation (which represents the marks for all students and all assignments) has been altered by this action. Any resources that previously were known to be current representations of the *assess* relation are now out of date. Only in a later step of the plan, when an old *assess* resource is combined with the output from the marking tool, is there once again a current resource for the entire relation *assess*.

3 Example Domain

The example used in this paper is from the domain of university course administration. At the author's 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. These tasks are often performed using a tool kit approach: the course administrator uses a number of different tools to perform the tasks, some being general-purpose tools he or she is familiar with, and some being specially written for work in this problem domain.

This paper will focus on a particular task from this domain: marking electronically-submitted student assignments. A simplified version of the relational model for the information structure of this domain is shown in Figure 2. 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.

4 Information Processing Operators and Methods

Figure 3 shows an example of an operator specification. This describes the invocation of a programming assignment marking utility that allows a tutor to systematically access student programs over a network, compile and run them and record a mark. The parameters are the assessment component being marked (e.g. Assignment 1), the set of student ID numbers whose

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 2: The relational model for the course administration domain

assignments should be marked (the class may be partitioned amongst several tutors), and the set of IDs for the students whose work actually was marked at the end of a session with the tool (it is not necessary to do all the marking in one session). This last parameter is represented by a run-time variable (indicated by the leading '!'). It will be instantiated once the action has been performed.

The box at the top of the figure has three compartments containing (respectively) the operator's name and parameters, the list of information state base relations affected by this operator, and a list of resource pre- and postconditions. The operator requires two resources. The first precondition resource corresponds to a selection on the domain's *student* relation and this resource is declared to still be valid after this action has been performed. A second precondition resource corresponds to the relation *assess*. This resource is declared to be invalidated by the action (because it changes the assess relation). A third resource is produced (but not required): this corresponds to a selection on the domain's *assess* relation. Sect. 5 gives the semantics of this notation in terms of the situation calculus [11].

The information state constraints describe how any relations affected by the action are modified. 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_some` operator, the first two constraints specify how the operator changes the relation `assess` (shown in diagrammatic form in Figure 4). 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

mark_some(Cmpt, ToMarkIDs, !MarkedIDs)
{assess}
$\sigma_{\text{student}}(\text{stu\_id} \in \text{ToMarkIDs}) \rightarrow$ assess $-\!\!\times$ $\quad \sigma_{\text{assess}}(\text{stu\_id} \in \text{!MarkedIDs} \wedge \text{cmpt} = \text{Cmpt})$

Information state constraints

var NewData: relation([stu_id: string, cmpt_id: string, mark: decimal(1)])

$\text{assess}' = \text{assess} \dot{\cup} \text{NewData}$

$\text{key\_values}(\text{NewData}, \text{stu\_cmpt}) = \text{!MarkedIDs} \times \{\text{Cmpt}\}$

$\text{!MarkedIDs} \subseteq \text{ToMarkIDs}$

Figure 3: An example operator specification

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}`.

Information state 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 as the knowledge in these constraints can be used to design *resource computation links* that describe how resources required in one state can be computed by combining existing resources from previous states.

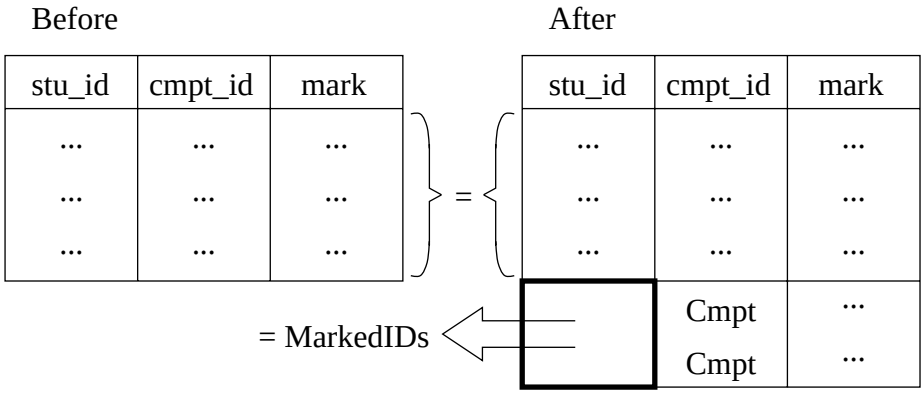


Figure 4: The effects of mark_some(Cmpt, ToMarkIDs, !MarkedIDs) on the relation assess

An example plan is shown in Figure 5. The representation extends that used in standard HTN planning by the addition of constraints on variables 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. To help show the flow of information in the figure, run-time variables are annotated with '!' in the task or constraint that instantiates them; the '!' is omitted in later references to that variable.

The first and second levels of this plan show how the task `mark(ass1)` has been expanded by its associated method into a `rel_tuples` action followed by a `mark_all` task. The `rel_tuples` action instantiates the run-time variable `!AllIDs` to a term representing the set of tuples in the relation $\pi_{student}(\{stu\_id\})$ (i.e. the set of all student IDs). The `mark_all` task requires a resource containing a subset of the *student* relation, with one tuple for each ID number in its argument `AllIDs`. The designer of this method knew that when this task follows the previous `rel_tuples` action, the value of `AllIDs` will ensure that the required information is in fact the entire *student* relation. This knowledge has been encoded in the method by the use of the '=' resource computation link (which in fact requires no computation — just the assertion in the execution environment of another record for the *student* resource, stating that it also corresponds in the current state to the relational expression *exp1* (with variable `AllIDs` instantiated).

The `mark_all` task has two methods. The one that has been used in the bottom of Fig. 5 expands a `mark_all` task into a `mark_some` action followed by a recursive call to the `mark_all` task. In addition a constraint is posted to ensure that the run-time variable `!RemainingIDs` will be instantiated by the execution environment once the run-time variable `!MarkedIDs` has been instantiated by the `mark_some` action. Knowledge from the information state constraints for `mark_some`'s operator has been encoded into this method by the two resource computation links. These state that the input resources for the recursive `mark_all` task can be computed from existing resources using a select operation and a (disjoint) union operation respectively.

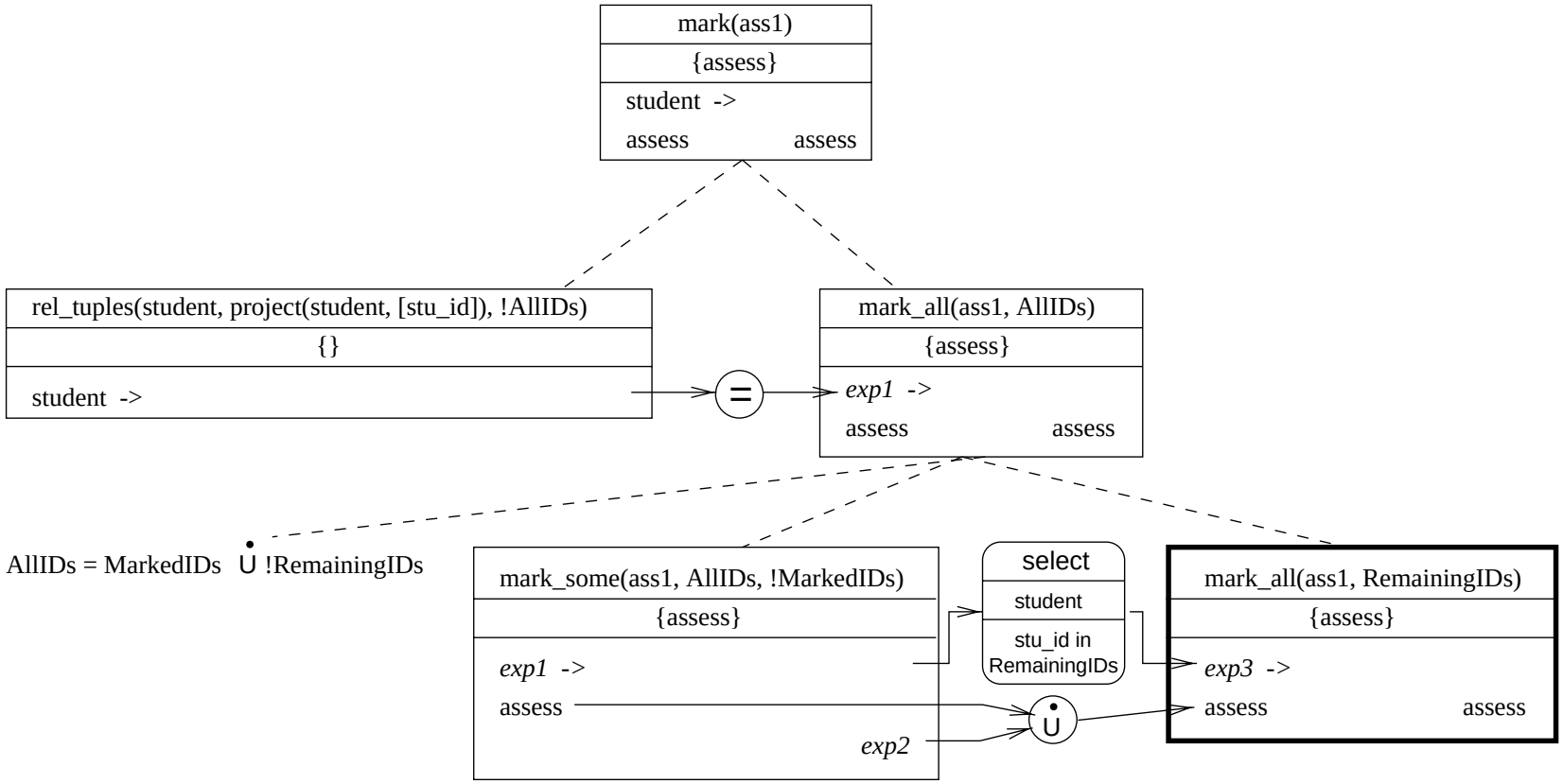
Future refinements of this plan will involve more expansions of `mark_all` tasks until finally its second parameter will be the empty set. In this case another method for `mark_all` will be used: one that expands into an empty set of tasks and asserts that its output resource for *assess* is the same as its input resource for this relation.

5 Semantics of Resource Pre- and Postconditions

This section presents the semantics of the notation used for operator specifications, as shown in Fig. 3. We use a variant of the situation calculus [11].

Consider a (possibly parameterised) operator *Op* with preconditions $\{Pre_1, \ldots, Pre_m\}$ and postconditions $\{Post_1, \ldots, Post_n\}$. The effects of *Op* are conditional on its preconditions being satisfied and can be expressed by the following situation calculus implication:

$$holds(Pre_1, S) \wedge \cdots \wedge holds(Pre_m, S) \rightarrow$$
$$holds(Post_1, result(Op, S)) \wedge \cdots \wedge holds(Post_n, result(Op, S))$$



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

Fig. 5. An example plan (the bold node is an unexpanded compound task)

where $holds(A, S)$ denotes that the atom A holds in state S and $result(Op, S)$ denotes the state resulting from performing the action represented by Op in state S.

The assertions we are particularly interested in as pre- and postconditions are those of the form:

$$valid\_resource(Res, RelExp, State)$$

and

$$invalid\_resource(Res, RelExp, State)$$

where Res is a resource locator (e.g. a URL) and $RelExp$ is a relational algebra expression. These two assertions state that the resource is valid (or, respectively, invalid) in the specified state for the given relation. The $invalid\_resource$ predicate is used to make assertions about resources that are *known* to be invalidated as the result of an action — there is a distinction between resources that are known to be invalid and those whose validity is not known. Therefore the postcondition that an action invalidates a resource is a strictly stronger statement than negating the validity of the resource, as the following axiom states:

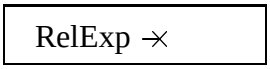
$$invalid\_resource(Res, RelExp, State) \rightarrow \neg valid\_resource(Res, RelExp, State)$$

Note that expressions denoting states appear within the pre and postconditions, not just as arguments to the *holds* meta-predicate. This is necessary because the task-expansion methods used in our planning framework allows resources from different states to be combined to produce new resources (see Fig. 5 where at the bottom of the figure, an out-of-date resource for the *assess* relation is merged with an up-to-date resource containing new *assess* tuples to produce a new resource for *assess*). For this to be possible, the execution environment must provide some facility for locating resources identified both in terms of their intellectual content (a relation) and also by a state for which the information is required. It must also support the generation of new state names as actions are performed, and keep a record of the name of the current state.

In the situation calculus (i.e. at the meta-level) the term denoting a state is constructed from the term denoting the prior state by using the *result* function. It is not necessary for this same scheme to be used at the object level provided that each state has a unique name. In the following we use the expression S'_{Op} to represent the state resulting from performing Op in state S.

The graphical representation of an operator specification (as shown in Fig. 3) contains the resource pre- and postconditions in the bottom compartment. Each line of this compartment can have one of the following forms, which have their semantics shown beside them. For a given operator, all the resource pre- and postconditions must be combined into a single situation calculus implication describing the operator's effects. Thus each of the forms shown below contributes one conjunct to the right hand side of this implication and (possibly) one conjunct to the left hand side. This is indicated by the use of "$\cdots$" below, which represents the conjuncts contributed by other resource pre- and postconditions.

RelExp $\rightarrow$	**A resource is required and preserved:**
	$valid\_resource(Res, RelExp, S) \wedge \cdots$ $\rightarrow valid\_resource(Res, RelExp, S'_{Op}) \wedge \cdots$



A resource is required and invalidated:

$$valid\_resource(Res, RelExp, S) \ \wedge \ \cdots \\ \rightarrow invalid\_resource(Res, RelExp, S'_{Op}) \ \wedge \ \cdots$$

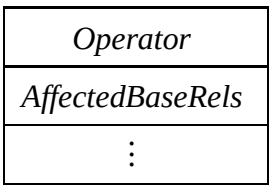
A resource is required and updated/replaced:

$$valid\_resource(Res, RelExp, S) \ \wedge \ \cdots \\ \rightarrow \exists Res' \ valid\_resource(Res', RelExp, S'_{Op}) \ \wedge \cdots$$

A new resource is generated:

$$\cdots \rightarrow \exists Res' \ valid\_resource(Res', RelExp, S'_{Op}) \wedge \cdots$$

In addition, an operator specifies which of the domain information model's base relations are affected by the action. This is used as a form of frame axiom: if a resource corresponds to a relational expression involving base relations that are not affected by the action, then that resource remains valid in the state following the action:



$$valid\_resource(Res, RelExp, S) \wedge \\ base\_relations\_in(RelExp, BR) \wedge \\ BR \cap AffectedBaseRels = \emptyset \\ \rightarrow valid\_resource(Res, RelExp, S'_{Op})$$

The execution environment is responsible for using the set of affected base relations to update the list of valid resources when it performs an action. This set can also be used by the execution environment to automatically delete $valid\_resource$ facts when an action is performed that is known to affect (and therefore possibly invalidate) a resource. However, this should only be done if the operator doesn't explicitly declare that the resource remains valid. This can be represented by the following axioms:

$$valid\_resource(Res, RelExp, S) \wedge \\ base\_relations\_in(RelExp, BR) \ \wedge \ BR \cap AffectedBaseRels \neq \emptyset \\ \rightarrow poss\_invalid\_resource(Res, RelExp, S'_{Op})$$

$$poss\_invalid\_resource(Res, RelExp, S) \ \wedge \ \neg valid\_resource(Res, RelExp, S) \\ \rightarrow \ invalid\_resource(Res, RelExp, S)$$

6 Implementation Issues

This planning scheme has been implemented by extending the Common Lisp HTN planner SHOP (Simple Hierarchical Order Planner) [12]. SHOP implements a simple form of HTN planning: plans are linearly ordered and tasks are planned for in the order in which they will be executed. This simplifies the treatment of task interactions (resulting in a small and easily understood implementation) and also allows methods to have preconditions that involve arbitrary Lisp computation.

SHOP's operators do not have preconditions — only add and delete lists of facts. However, its methods do have preconditions. Therefore, our operators are encoded as methods with preconditions. These methods expand into an associated operator which has all the arguments of the method as well as arguments for any free variables in the preconditions. The operator then asserts and retracts $valid\_resource$ and $invalid\_resource$ constraints. These 'operator' methods, as well as ordinary methods, have arguments representing the resource locators for the input and output resources. These allow methods to represent the flow of information between subtasks by using the same variable for an output resource argument of one task and an input resource argument for another task.

It was necessary to add to each of our operators a precondition to fetch the name of the prior state and a postcondition to assert the name of the resulting state. This was because the names of these two states appears in the other pre- and postconditions.

It was also necessary to extend SHOP to support interleaved planning and execution and run-time variables. This has been done in a simple fashion at present: when the planner evaluates a method, after the preconditions have been checked, the user is asked to simulate the execution of this action by providing values for any of its parameters that were declared as run-time variables. To integrate the planner with the rest of our agent architecture, rather than pausing the planner to await the values of run-time variables when an action is to be performed, it will be more scalable for the planner to terminate with the current partial plan and accept a later request to further refine a more instantiated version of the partial plan.

Resource computation links are implemented by methods and associated operators with a run-time variable representing the resource locator for the newly created resource. Resources created by actions are also represented by run-time variables.

One aspect of our framework that was difficult to implement in a general in SHOP was allowing the posting of constraints in methods — this would be better supported by a planner based on a constraint logic programming language. For the example in this paper a special 'set subtraction' task was inserted between the two subtasks in the `mark_all` task expansion. The method for this task uses its preconditions to compute the resulting set — this involved using SHOP's facility for providing Horn clauses that are checked when evaluating preconditions.

7 Related work

There are a number of planners designed to plan for gathering information from large dynamic networks of information sources ([3–6]), but none of these are designed with information pro-

cessing tasks specifically in mind.

XII [3] 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 [4], the planner used in the SIMS project [13], 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 [5] is an special-purpose algorithm designed for the same task as Sage. It models the available information 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.* [6] 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. This allows information processing 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. Instead, iterative behaviour can be achieved through the use of run-time variables and recursive task-expansion methods.

8 Conclusion

This paper has presented a variant of HTN planning designed for supporting the interoperation of information processing tools. Novel features include the explicit distinction between the information state effects of an action and its resource pre- and postconditions, a solution to the frame problem based on a declaration of the set of base relations affected by each operator, and the use of resource computation links in methods to show how existing resources can be combined to form new resources required by tasks. The model has been tested by encoding it in an extended version of an existing HTN planner.

References

[1] Object Management Group. OMG homepage. http://www.omg.org/.

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

[3] 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.

[4] 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.

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

[6] 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.

[7] S. Cranefield, E. Moreale, B. McKinlay, and M. Purvis. Automating the interoperation of information processing tools. In *Proceedings of the 32nd Hawaii International Conference on System Sciences (HICSS-32)*. IEEE, 1999. (CDROM, 10 pages).

[8] 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.

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

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

[11] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4. Edinburgh University Press, 1969. Reprinted in J. Allen, J. Hendler and A. Tate, Readings in Planning, Morgan Kaufmann, 1990.

[12] D. Nau, Y. Cao, A. Lotem, and H. Munõz-Avila. SHOP: Simple hierarchical ordered planner. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence(IJCAI-99)*, 1999. To appear.

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