Planning in a Multi-Agent Environment: Theory and Practice
ID number: 270

[Extended Abstract]

Jürgen Dix[*]
The University of Manchester
Dept. of CS, Oxford Road
Manchester M13 9PL, UK
dix@cs.man.ac.uk

Héctor Muñoz-Avila
Dept. of CSE
Lehigh University
USA
munoz@eecs.lehigh.edu

Dana Nau/Lingling Zhang
University of Maryland
Dept. of CS
College Park, MD 20752, USA
{nau,lingl}@cs.umd.edu

ABSTRACT

We give the theoretical foundations and empirical evaluation of a planning agent, `shop`, performing *HTN* planning in a multi-agent environment. `shop` is based on *A-SHOP*, an agentized version of the original *SHOP HTN* planning algorithm, and is integrated in the *IMPACT* multi-agent environment. We run several experiments involving accessing various distributed, heterogeneous information sources, based on simplified versions of noncombatant evacuation operations, NEO's. As a result, we noticed that in such realistic settings the time spent on communication (including network time) is orders of magnitude higher than the actual inference process. This has important consequences for optimizations of such planners. Our main results are: (1) using NEO's as new, more realistic benchmarks for planners acting in an agent environment, and (2) a memoization mechanism implemented on top of `shop`, which improves the overall performance in a significant way.

Categories and Subject Descriptors

I.2.12 [**Artificial Intelligence**]: Distributed AI—*Intelligent Agents*

Keywords

Agent architectures, Agent selection and planning, Formalisms and logics

1. INTRODUCTION

Planning a course of action is difficult, especially for large military organizations (e.g., the U.S. Navy) that have their available assets distributed world-wide. Formulating a plan in this context requires accessing remote, heterogeneous information sources. For example, when planning for a *Noncombatant evacuation operation*, denoted by NEO, military commanders must access several information sources including: assets available in the zone of operations, Intelligence assessment about potential hostiles, weather conditions and so forth.

A-SHOP is an *HTN* planning algorithm for planning in a multi-agent environment. *A-SHOP* can interact with external information sources, frequently heterogeneous and not necessarily centralized, via the *IMPACT* multi-agent environment. The *IMPACT* project (see [7, 22] and `http://www.cs.umd.edu/projects/impact/`) aims at developing a powerful and flexible, yet easy to handle framework for the interoperability of distributed heterogeneous sources of information.

In previous work we described the definition of the *A-SHOP* planning algorithm, an agentized version of *SHOP* that runs in the *IMPACT* environment and formulated the conditions needed for *A-SHOP* to be sound and complete [6].

In this paper we will focus on the actual implementation of *A-SHOP* following the principles stated in our previous work and experiments we did on a transportation domain for NEO operations. Our analysis of the initial runs of *A-SHOP* revealed that most of the running time was spent on communication between the *IMPACT* agents and accessing the information sources. Compared to that, the actual inferencing time in *A-SHOP* was very small. Furthermore, we observed that frequently the same *IMPACT* query was performed several times. To solve this problem we implemented a memoization mechanism to avoid repeating the same *IMPACT* queries. As we will show, the key for this mechanism to work is that the *A-SHOP* algorithm performs a planning technique called ordered task decomposition. As a result, *A-SHOP* maintains partial information about the state of the world. Experiments performed show that the memoization mechanism results in a significant reduction of the running time in *A-SHOP*. This reduction depends on the overall network time spent to access the information sources: the higher this network time, the higher is the gain obtained by our memoization technique.

---

[*] Responsible Author.

AAMAS '02 Bologna, Italy

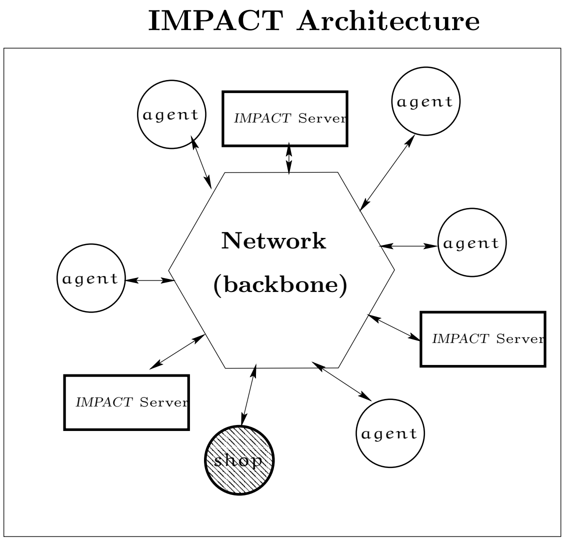

Figure 1: *SHOP* as a planning agent in *IMPACT*.

This paper is organized as follows. The next section describes the Noncombatant evacuation operations (NEO's) planning domain, which partly motivated our approach. In Section 3.2 we introduce *IMPACT*, define*A-SHOP* and the results establishing the soundness and completeness of *A-SHOP*. Section 4 describes the actual implementation of *A-SHOP*. Section 5 describes the memoization mechanism and its dependence on the Ordered Task Decomposition planning technique. In Section 6 we describe several experiments with *A-SHOP* for logistics NEO problems. Finally, we discuss related work in Section 7 and conclude with Section 8.

2. PLANNING NONCOMBATANT EVACUATION OPERATIONS (NEO'S)

Noncombatant evacuation operations are conducted to assist the U.S.A. Department of State (*DOS*) with evacuating noncombatants, nonessential military personnel, selected host-nation citizens, and third country nationals whose lives are in danger from locations in a host foreign nation to an appropriate safe haven. They usually involve the swift insertion of a force, temporary occupation of an objective (e.g., an embassy), and a planned withdrawal after mission completion. NEO's are often planned and executed by a Joint Task Force (*JTF*), a hierarchical multi-service military organization, and conducted under an American Ambassador's authority. Force sizes can range into the hundreds and involve all branches of the armed services, while the evacuees can number into the thousands. More than ten NEO's were conducted within the past decade. Publications describe NEO doctrine [17], case studies [21], and more general analysis (e.g., [20]).[1]

The decision making process for a NEO is conducted at three increasingly-specific levels: *strategic*, *operational* and *tactical*. The strategic level involves global and political considerations such as whether to perform the NEO. The operational level involves considerations such as determining the size and composition of its execution force. The tactical level is the concrete level, which assigns specific resources to specific tasks. Thus, this domain is particularly suitable for a hierarchical (*HTN*) planning approach.

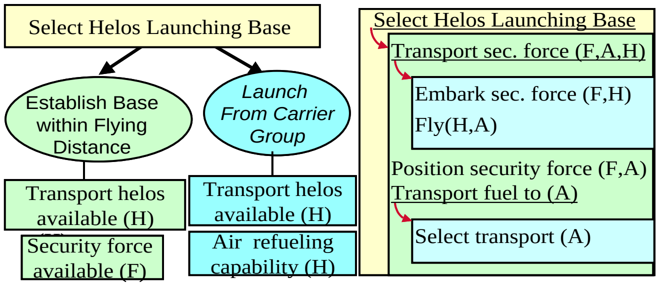
Select Helos Launching Base
Establish Base within Flying Distance
Launch From Carrier Group
Transport helos available (H)
Security force available (F)
Transport helos available (H)
Air refueling capability (H)
Select Helos Launching Base
Transport sec. force (F,A,H)
Embark sec. force (F,H)
Fly(H,A)
Position security force (F,A)
Transport fuel to (A)
Select transport (A)

Figure 2: NEO transportation example.

JTF commanders plan NEO's by gathering information from multiple sources. For example, in preparation for *Operation Eastern Exit* (Mogadishu, Somalia, 1991), commanders accessed Intelligence Satellite Photographs from NIMA (National Imagery and Mapping Agency), intelligence assessment information from the *CIA*, the Emergency Action Plan (*EAP*) from the US Embassy in Mogadishu, among others [21]. Any automated system planning in this domain must be able to access these multiple distributed information sources.

3. PLANNING WITH REMOTE, HETEROGENEOUS INFORMATION SOURCES

In this section we review results obtained in [6]. After giving a brief overview on *SHOP* and *IMPACT* in Subsection 3.1, we state the main results of [6] in Subsection 3.4.

3.1 SHOP

Rather than giving a detailed description of the kind of *HTN* planning used by *SHOP* ([16]), we consider the following example.

In order to do planning in a given planning domain, *SHOP* needs to be given knowledge about that domain. *SHOP*'s knowledge base contains *operators* and *methods*. Each operator is a description of what needs to be done to accomplish some primitive task, and each method is a prescription for how to decompose some complex task into a totally ordered sequence of subtasks, along with various restrictions that must be satisfied in order for the method to be applicable.

Given the next task to accomplish, *SHOP* chooses an applicable method, instantiates it to decompose the task into subtasks, and then chooses and instantiates other methods to decompose the subtasks even further. If the constraints on the subtasks prevent the plan from being feasible, *SHOP* will backtrack and try other methods.

As an example, Figure 2 shows two methods for the task of *selecting a helicopter launching base*: *establishing the base within flying distance*, and *launch from carrier battle group* (i.e., use the carrier as the helicopter launching base). Note that each method's preconditions are not used to create subgoals (as would be done in action-based planning). Rather, they are used to determine whether or not the method is applicable. *Establishing the base within flying distance* requires to have transport helicopters and a security force available. Launching from carrier battle group also requires to have helicopters available and those helicopters have to have air refuelling capability (which wasn't necessary in the first method because the helicopters are within flying dis-

---

[1] See `www.aic.nrl.navy.mil/~aha/neos` for more information on NEO's.

tance).

If the method *establishing base within flying distance method* is selected, the select helicopter launching base is decomposed into three subtasks: transport security force (F) using the helicopters (H) to the selected launching base (A), position the security force in the base, and transport the fuel to the base. Some of these tasks, such as transporting the security force, can be further decomposed. Others such as *position security force* cannot. The former are called *compound* tasks, the latter *primitive* tasks.

Here are some of the complications that can arise during the planning process:

- The planner may need to recognize and resolve interactions among the subtasks. For example, in planning how to travel to the airport, one needs to make sure that the fuel arrive after the security force has been deployed. To make the example in Figure 2 more realistic, such information would need to be specified as part of *SHOP*'s methods and operators.
- In the example in Figure 2, it was always obvious which method to use. But in general, more than one method may be applicable to a task. If it is not possible to solve the subtasks produced by one method, *SHOP* will backtrack and try another method instead.

3.2 IMPACT

To get a bird's eye view of *IMPACT*, here are the most important features:

Actions: Each *IMPACT* agent has certain *actions* available. Agents act in their environment according to their *agent program* and a well defined *semantics* determining which of the actions the agent should execute.

Cycle: Each agent undergoes the following cycle:

(1) Get messages by other agents. This changes the state of the agent.
(2) Determine (based on its program, its semantics and its state) for each action its *status* (permitted, obliged, forbidden, ...). The agent ends up with a *set of status atoms.*
(3) Based on a notion of concurrency, determine the actions that can be executed and update the state accordingly.

Legacy Code: *IMPACT* Agents are built on top of arbitrary software code (*Legacy Data*).

Agentization: A methodology for transforming legacy code into an *agent* has been developed.

A complete description of all these notions is out of scope of this paper and we refer to [22] for a detailed presentation.

Before explaining an agent in more detail, we need to make some comments about the general architecture. In *IMPACT* agents communicate with other agents through the network. Not only can they send out (and receive) messages from other agents, they can also ask the server to find out about services that other agents offer. For example a planning agent (we call it *A-SHOP*), confronted with a particular planning problem, can find out if there are agents out there with the data needed to solve the planning problem; or agents can provide *A-SHOP* with information about relevant legacy data see Figure 1.

One of the main features of *IMPACT* is to provide a method (see [22]) for *agentizing* arbitrary legacy code, i.e. to turn such legacy code into an agent. In order to do this, we need to abstract from the given code and describe its main features. Such an abstraction is given by the set of all datatypes and functions the software is managing. We call this a *body of software code* and denote it by $\mathcal{S} = (\mathcal{T}_{\mathcal{S}}, \mathcal{F}_{\mathcal{S}})$. $\mathcal{F}_{\mathcal{S}}$ is a set of predefined functions which makes access to the data objects managed by the agent available to external processes.

For example, in many applications a $\mathtt{math}$ agent is needed. This agent is able to do mathematical calculations shipped to it by other agents. For example it can determine the time it takes for a particular vehicle to get from one location to another. Another example is a $\mathtt{monitoring}$ agent, that keeps track of distances between two given points and the authorized range or capacity of certain vehicles. These information can be stored in several databases.

Definition 3.1 (State of an Agent, $\mathcal{O}_{\mathcal{S}}(t)$) *At any given point t in time, the* state of an agent*, denoted $\mathcal{O}_{\mathcal{S}}(t)$, is the set of all data objects that are currently stored in the relations the agent handles—the types of these objects must be in the base set of types in $\mathcal{T}_{\mathcal{S}}$.*

In the example just mentioned, the state of the $\mathtt{monitoring}$ agent consists of all tuples stored in the databases it handles.

3.2.1 The Code Call Machinery

To perform logical reasoning on top of third party data structures (which are part of the agent's state) and code, the agent must have a language within which it can reason about the agent state. We therefore introduce the concept of a *code call atom*, which is the basic syntactic object used to access multiple heterogeneous data sources.

Definition 3.2 (Code Calls (cc)) *Suppose $\mathcal{S} =_{def} (\mathcal{T}_{\mathcal{S}}, \mathcal{F}_{\mathcal{S}})$ is some software code, $f \in \mathcal{F}_{\mathcal{S}}$ is a predefined function with n arguments, and $\mathtt{d}_1, \ldots, \mathtt{d}_\mathtt{n}$ are objects or variables such that each $\mathtt{d}_\mathtt{i}$ respects the type requirements of the i'th argument of f. Then, $\mathcal{S} : f(\mathtt{d}_1, \ldots, \mathtt{d}_\mathtt{n})$ is a* code call*. A code call is* ground *if all the $\mathtt{d}_\mathtt{i}$'s are objects.*

We often identify software code $\mathcal{S}$ with the agent that is built on top of it. This is because an agent really is uniquely determined by it.

A code call executes an *API* function and returns as output a set of objects of the appropriate output type. Going back to our agent introduced above, $\mathtt{monitoring}$ may be able to execute the cc $\mathtt{monitoring} : \mathit{distance}(\mathtt{locFrom}, \mathtt{locTo})$. The $\mathtt{math}$ agent may want to execute the following code call: $\mathtt{math} : \mathit{computeTime}(\mathtt{cargoPlane}, \mathtt{locFrom}, \mathtt{locTo})$.

What we really need to know is if the result of evaluating such code calls is contained in a certain set or not. To do this, we introduce code call atoms. These are *logical atoms* that are layered on top of code calls. They are defined through the following inductive definition.

Definition 3.3 (Code Call Atoms ($\mathbf{in}(\mathtt{X}, \mathtt{cc})$)) *If* `cc` *is a code call, and* `X` *is either a variable symbol, or an object of* *the output type of* $\mathtt{cc}$*, then* $\mathbf{in}(\mathtt{X}, \mathtt{cc})$ *and* $\mathbf{not\_in}(\mathtt{X}, \mathtt{cc})$ *are* code call atoms. $\mathbf{not\_in}(\mathtt{X}, \mathtt{cc})$ *succeeds if X is **not** in the set of objects returned by the code call cc.*

Code call atoms, when evaluated, return boolean values, and thus may be thought of as special types of logical atoms. Intuitively, a code call atom of the form $\mathbf{in}(\mathtt{X}, \mathtt{cc})$ succeeds if $\mathtt{X}$ can be set to a pointer to one of the objects in the set of objects returned by executing the code call.

As an example, the following code call atom tells us that the particular plane "$f22$" is available as a cargo plane at *ISB1*: $\mathbf{in}(f22, \mathsf{transportAuthority} : cargoPlane(\mathtt{ISB1}))$

Often, the results of evaluating code calls give us back certain values that we can compare. Based on such comparisons, certain actions might be fired or not. To this end, we need to define *code call conditions*. Intuitively, a code call condition is a conjunction of code call atoms, equalities, and inequalities. Equalities, and inequalities can be seen as additional syntax that "links" together variables occurring in the atomic code calls.

Definition 3.4 (Code Call Conditions (ccc))

1. *Every code call atom is a code call condition.*
2. *If* $\mathtt{s}, \mathtt{t}$ *are either variables or objects, then* $\mathtt{s} = \mathtt{t}$ *is a code call condition.*
3. *If* $\mathtt{s}, \mathtt{t}$ *are either integer/real valued objects, or are variables over the integers/reals, then* $\mathtt{s} < \mathtt{t}, \mathtt{s} > \mathtt{t}, \mathtt{s} \geq \mathtt{t}, \mathtt{s} \leq \mathtt{t}$ *are code call conditions.*
4. *If* χ_1, χ_2 *are code call conditions, then* $\chi_1 \,\&\, \chi_2$ *is a code call condition.*

A code call condition satisfying any of the first three criteria above is an atomic *code call condition.*

3.3 Agent Programs and Semantics

We are now coming to the very heart of the definition of an agent: its *agent program*. Such a program consists of rules of the form:

$$\mathsf{Op}\alpha(t_1,\ldots,t_m) \quad\leftarrow\quad \mathsf{Op}_1\beta_1(\ldots),\ldots,\mathsf{Op}_n\beta_n(\ldots), ccc_1,\ldots,ccc_r,$$

where $\alpha, \beta_1, \ldots \beta_n$ are *actions*, $\mathsf{Op}_1, \ldots, \mathsf{Op}_n$ describe the status of the action (*obliged*, *forbidden*, *waived*, *doable*) and ccc_i are code call conditions to be evaluated in the actual state.

Thus, Op_i are operators that take actions as arguments. They describe the status of the arguments they take. Here are some examples of actions: (1) to load some cargo from a certain location, (2) to fly a plane from a certain location to another location, (3) to unload some cargo from a certain location. The action status atom **F***load* (resp. **Do** *fly*) means that the action *load* is forbidden (resp. *fly* should be done). Actions themselves are terms, only with an operator in front of them they become atoms.

In *IMPACT*, actions are very much like STRIPS operators: they have preconditions and add and delete-lists (see appendix). The difference to STRIPS is that these preconditions and lists consist of *arbitrary code call conditions*, not just of logical atoms.

Figure 3 illustrates that the agent program together with the chosen semantics SEM and the state of the agent determines the set of all status atoms. However, the doable actions among them might be conflicting and therefore we have to use the chosen concurrency notion to finally determine which actions can be concurrently executed. The agent then executes these actions and changes its state.

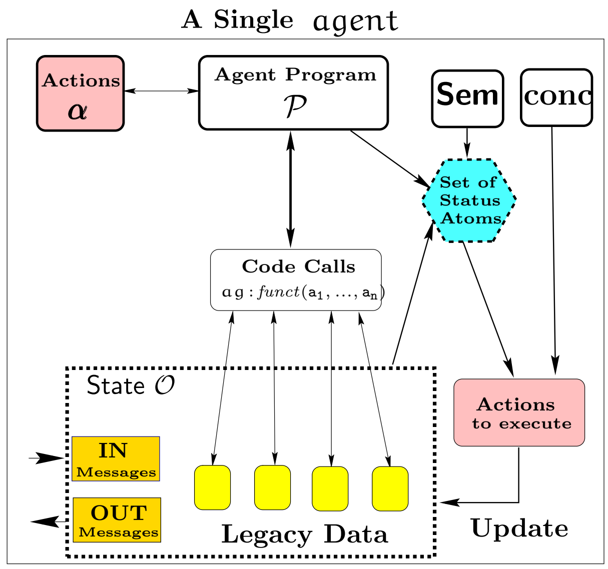

Figure 3: An Agent in *IMPACT*.

3.4 IMPACTING SHOP

A comparison between *IMPACT*'s actions and *SHOP*'s methods shows that *IMPACT* actions correspond to fully instantiated methods, i.e. no subtasks. While *SHOP*'s methods and operators are based on *STRIPS*, the first step is to modify the atoms in *SHOP*'s preconditions and effects, so that *SHOP*'s preconditions will be evaluated by *IMPACT*'s code call mechanism and the effects will change the state of the *IMPACT* agents. This is a fundamental change in the representation of *SHOP*. In particular, it requires replacing *SHOP*'s methods and operators with *agentized* methods and operators. These are defined as follows.

Definition 3.5 (Agentized Meth.: (AgentMeth $h\,\chi\,$t)) *An* agentized method *is an expression* (**AgentMeth** $h\,\chi\,$**t**) *where h (the method's* head*) is a compound task, χ (the method's* preconditions*) is a code call condition and* **t** *is a totally ordered list of subtasks, called the* task list.

The primary difference between definition of an agentized method and the definition of a method in *SHOP* is as follows. In *SHOP*, preconditions were logical atoms, and *SHOP* would infer these preconditions from its current state of the world using Horn-clause inference. In contrast, the preconditions in an agentized method are *IMPACT*'s code call conditions rather than logical atoms, and *A-SHOP* (the agentized version of *SHOP* defined in the next section) does not use Horn-clause inference to establish these preconditions but instead simply invokes those code calls, which are calls to other agents (which may be Horn-clause theorem provers or may instead be something entirely different).

Definition 3.6 (Agentized Op.: (AgentOp $h\,\chi_{add}\,\chi_{del}$)) *An* agentized operator *is an expression* (**AgentOp** $h\,\chi_{add}\,\chi_{del}$)*, where h (the* head*) is a primitive task and χ_{add} and χ_{del} are lists of code calls (called the* add- *and* delete-lists*). The set* *of variables in the tasks in* χ_{add} *and* χ_{del} *is a subset of the set of variables in* h*.*

The Algorithm

```
procedure A-SHOP($\mathbf{t}, \mathcal{D}$)
  1. if $\mathbf{t} = nil$ then return $nil$
  2. $t :=$ the first task in $\mathbf{t}$; $R :=$ the remaining tasks
  3. if $t$ is primitive and a simple plan for $t$ exists then
  4.     $q := simplePlan(t)$
  5.     return $concatenate(q, A\text{-}SHOP(R, \mathcal{D}))$
  6. else if $t$ is non-prim. $\wedge$ there is a reduction of $t$ then
  7.     nondeterministically choose a reduction:
         Nondeterministically choose an agentized method,
          (AgentMeth $h\,\chi\,t$), with $\mu$ the most general
          unifier of $h$ and $t$ and substitution $\theta$ s.t.
          $\chi\mu\theta$ is ground and holds in IMPACT's state $\mathcal{O}$.
  8.     return $A\text{-}SHOP(concatenate(t\mu\theta, R), \mathcal{D})$
  9. else return FAIL
 10. end if
end A-SHOP

procedure simplePlan($t$)
 11. nondeterministically choose agent. operator
     $\mathsf{Op} =$ (AgentOp $h\,\chi_{add}\,\chi_{del}$) with $\nu$ the most
     general unifier of $h$ and $t$ s.t. $h$ is ground
 12. monitoring : $apply(\mathsf{Op}\,\nu)$
 13. return $\mathsf{Op}\,\nu$
end A-SHOP
```

Figure 4: *A-SHOP*, the agentized version of *SHOP*.

The *A-SHOP* algorithm is now an easy adaptation of the original *SHOP* algorithm. Unlike *SHOP* (which would apply an operator by directly inserting and deleting atoms from an internally-maintained state of the world), *A-SHOP* needs to reason about how the code calls in an operator will affect the states of other agents. One might think the simplest way to do this would be simply to tell these agents to execute the code calls and then observe the results, but this would not work correctly. Once the planning process has ended successfully, *A-SHOP* will return a plan whose operators can be applied to modify the states of the other *IMPACT* agents—but *A-SHOP* should not change the states of those agents during its planning process because this would prevent *A-SHOP* from backtracking and trying other operators.

Thus in Step 12, *SHOP* does not issue code calls to the other agents directly, but instead communicates them to a `monitoring` agent. The `monitoring` agent keeps track of all operators that are supposed to be applied, without actually modifying the states of the other *IMPACT* agents. When *A-SHOP* queries for a code call $cc = \mathcal{S} : f(\mathtt{d}_1, \dots, \mathtt{d}_n)$ in χ to evaluate a method's precondition (Step 7), the monitoring agent examines if *cc* has been affected by the intended modifications of the operators and, if so, it evaluates *cc*. If *cc* is not affected by application of operations, *IMPACT* evaluates *cc* (i.e., by accessing $\mathcal{S}$). The list of operators maintained by the monitoring agent is reset whenever a planning process begins. The *apply* function applies the operators and creates copies of the state of the world. Depending on the underlying software code, these changes might be easily revertible or not. In the latter case, the monitoring agent has to keep track of the old state of the world.

3.5 Finite Evaluability of ccc's and Completeness of ASHOP

An important question for any planning algorithm is whether all solution plans produced by the algorithm are correct (i.e., soundness of the algorithm) and whether the algorithm will find solutions for solvable problems (i.e., completeness of the algorithm). Soundness and completeness proofs of classical planners assume that the preconditions can be evaluated relative to the current state. In *SHOP*, for example, the state is accessed to test whether a method is applicable, by examining whether the method's preconditions are valid in the current state. Normally it is easy to guarantee the ability to evaluate preconditions, because the states typically are lists of predicates that are locally accessible to the planner. However, if these lists of predicates are replaced by code call conditions, this is no longer the case.

Code call conditions provide a simple, but powerful language syntax to access heterogeneous data structures and legacy software code. However, in general their use in agent programs is not limited. In particular, it is possible that a ccc can not be evaluated (and thus the status of actions can not be determined) simply because there are uninstantiated variables and thus the underlying functions can not be executed.

We have introduced syntactic conditions, similar to *safety* (and consequently called *strong safety*) in classical databases, to ensure evaluability and termination of ccc's (see [8, 22]).

Lemma 3.7 (Evaluating Agentized Operators) *Let* (**AgentMeth** $h\,\chi\,\mathbf{t}$) *an agentized method,* $\mathcal{O}$ *a state, and* (**AgentOp** $h'\,\chi_{add}\,\chi_{del}$) *an agentized operator. If the precondition* χ *is strongly safe wrt. the variables in* h*, the problem of deciding whether* χ *holds in* $\mathcal{O}$ *can be algorithmically solved. If the add and delete-lists* χ_{add} *and* χ_{del} *are strongly safe wrt. the variables in* h'*, the problem of applying the agentized operator to* $\mathcal{O}$ *can be algorithmically solved.*

Theorem 3.8 (Soundness, Completeness) *Let* $\mathcal{O}$ *be a state and* $\mathcal{D}$ *be a collection of agentized methods and operators. If all the preconditions in the agentized methods and add and delete-lists in the agentized operators are strongly safe wrt. the respective variables in the heads, then A-SHOP is correct and complete.*

4. ASHOP: IMPLEMENTATION

Each cycle in the *A-SHOP* algorithm consist of three phases (see lines 3 and 7 of Figure 3.4):

1. *Selection Phase*: Selecting a candidate agentized method or operator to reduce a task.
2. *Evaluation Phase*: Evaluating the applicability of the chosen agentized method or operator.
3. *Reduction Phase*: Performing the agentized method or operator.

To accomplish these phases we have implemented 3 *IMPACT* agents which perform pieces of these phases:

`ashop`**:** This is the agent that all *IMPACT* agents communicate with for generating a plan. It receives as input a problem and outputs a solution plan. The *A-SHOP* agent also performs the Selection Phase and the evaluation phase for the situation in which an operator is chosen. The operator is then send to the Monitor Agent, to perform a *virtual execution* of it. If the selection of a method is made, the *A-SHOP* agent sends a message to the Preconditions Agent with the code-call condition of the selected method.

`preconditions`: Receives a code-call condition and evaluates each code-call by sending it to the Monitoring Agent.

`monitoring`: The monitor agent has two functions: firstly, it receives a operator and performs a virtual execution of it. Secondly, it receives code-calls and evaluates them. We explain both of these operations in detail below as they are closely inter-related.

One of the main issues we are confronted with during the implementation is how to cope with the *execution* of agentized operators. In classical AI planning, where the state is centralized, executing an operator is a matter of simply making the changes to the state indicated by the operator and keeping track of those changes in an stack; if backtracking occurs, the stack is used to restore to the previous state.

This approach is not working in a multi-agent environment, where the state is distributed among several information sources (see Definition 3.1). Firstly, remote information sources might not be able to backtrack to a previous state. Secondly, even if backtracking was possible, performing such an operation may be costly. Thirdly, executing an operation may make resources unavailable temporarily for other agents and if backtracking takes place, these resources could have been used. For example, an operator may reserve a *recon* plane but a later operator trying to provide flight escort to the *recon* plane might not succeed. In this case the original *recon* plane should have not been reserved in the first place.

The Monitoring Agent overcomes these problems by keeping track of each operator execution without accessing the corresponding information sources to request an execution of the operation. For this reason we refer to this as a *virtual operator execution*. Since `monitoring` keeps track of the changes in the states of the remote information sources, the `preconditions` sends the code-calls to the `monitoring`. `monitoring` makes the code-call to the corresponding information source and then checks if the answer is affected by the previously virtually executed operators before sending its answer to the `preconditions`. For example, in a previous operator execution a *recon* plane might have been reserved. As port of the same planning episode, a code-call may enquire if there are any *recon* planes available. `monitoring` receives this code call and passes to the corresponding information source. Once the information source returns the answer (there is one *recon* plane available), `monitoring` checks the virtually executed list and discovers that for an operator has locked the *recon* plane. Thus, `monitoring` returns to the Precondition Agent that this code-call cannot be satisfied.

When the plan is completed, the actual execution is done in two runs: in the first run all resources used by the plan are locked by informing the corresponding information sources. This step is necessary to make sure that the resources are still available. If all resources are available, in the second step the plan is actually executed. That is, each of the agentized operators in the plan is applied and the states of the corresponding remote information sources are changed. If the first step fails because a resource is no longer available, *A-SHOP* will need to generate a new plan. An alternative approach, could be for the monitoring agent to lock the resources during the planning process. This avoids the need to make the two runs at the end but will make the resources unavailable as explained before. In future work, we plan to study the trade-offs between these alternatives.

5. MEMOIZATION IN ASHOP

While our implementation secures that the produced plans are consistent, the resulting running time was large compared to the inferencing time (we will describe the experiments later). Our experiments show that the bulk of the planning time has been spent in accessing the remote information sources. Further analysis revealed that the same code-calls were repeatedly being executed during the planning process. Our solution was to implement a cache mechanism to avoid repeated evaluations of the same code call in *IMPACT*.

Again this issue marked a difference from classical AI planning approaches. In *SHOP*, for example, we use a hash table to quickly check the validity of a condition in the current state. Other planning systems use more sophisticated data structures to reduce the time for evaluating a condition in the current state. For example, *TLPlan*, the winner of the 2000 AI planning competition, uses a bit map that allows checking conditions in almost constant time [1].

Obviously none of these techniques would be useful here since the information sources are remote and *A-SHOP* has no control over how data is stored there and how it is updated. However, implementing a memoization mechanism turned out to be adequate for *A-SHOP* for two reasons: Firstly, *A-SHOP* performs *Ordered Task Decomposition.* Secondly, all access to the information sources is canalized through `monitoring`.

As explained above, *A-SHOP* maintains a totally ordered list of tasks and starts planning for the first one, then the second one and so on. This technique is called *Ordered Task Decomposition.* A direct consequence of this technique is that *A-SHOP* can maintain partial information about the state of the world. In particular, code-calls that are made form the partial information about the state of the world and the sequence of virtually executed operators indicate the changes in that partial state.

The fact that access to the information sources is canalized through `monitoring` makes this agent the natural place for maintaining the updated partial state of the world. As a result, we modified `monitoring` as follows:

- When it receives a code-call from `preconditions`, the `monitoring` will first check if the code-call can be answered based on previous code-calls and the modifications indicated by the virtually executed operators. Only if it is not possible to answer this code call, the remote information source is accessed via the *IMPACT* code-call evaluation mechanism.
- After, receiving the answer from *IMPACT* for the evaluation of the code-call, `monitoring` records this answer.

In the example of the *recon* plane, after the first operator reserving the *recon* plane is virtually executed, `monitoring` knows that there are no more *recon* planes available. Thus, as it receives the code-call enquiring about the availability of *recon* planes it will answer that this code-call cannot be satisfied without having to access the corresponding remote information source via *IMPACT*. As will be shown next, these changes resulted in a reduction of the running time.

6. EMPIRICAL EVALUATION

The test domain is a simple transportation planning for a NEO [15]. Its plans involve performing a rescue mission where troops are grouped and transported between an initial location (the assembly point) and the NEO site (where the evacuees are located). After the troops arrived at the NEO site, evacuees are re-located to a safe haven.

Planning involves selecting possible pre-defined routes, consisting of four or more segments each. The planner must also choose a transportation mode for each segment. In addition, other conditions were determined during planning such as whether communication exists with *State Department personnel* and the type of evacuee registration process. *A-SHOP*'s knowledge base included six agentized operators and 22 agentized methods. There were four *IMPACT* information sources available:

- Transport Authority: Maintains information about the transportation assets available at different locations.
- Weather Authority: Maintains information about the weather conditions at the different locations.
- Airport Authority: Maintains information about availability and conditions of airports at different locations.
- Math Agent: `math` evaluates arithmetic expressions. typical evaluations include the subtract a certain number of assets use for an operation and update time delays.

The top level task for each problem in this experiment was the following: to perform a troop insertion and evacuees extraction plan. This top level task is decomposed into several subtasks, one for each segment in the route that the troops must cover (these segments are pre-determined as part of the problem description). Within each segment, *A-SHOP* must plan for the means of transportation (planes, helicopters, vehicles etc.) to be used and select a route for that segment. The selection of the means of transportation depends on their availability for that segment, the weather conditions, and, in the case of airplanes, on the availability and conditions of airports. The selection of the route depends on the transportation vehicle used and may lead to backtracking. For example, the choice of ground transportation assets needs to be revised if no roads are available or they are blocked, or too risky to take.

We ran our experiments on 30 problems of increasing size. The first five problems had four segments passing over five locations (including a particular location known as the Intermediate Staging Base *ISB*), the next five problems had five segments passing over six locations (two *ISB*'s), and so forth until the Problems 26–30 which had nine segments passing over 10 locations (five *ISB*'s).

We ran shop in two modes: with and without the memoization mechanism and measured for each mode two variables: *inferencing time* and *total time*. The inferencing time includes the time spent in the three agents implementing the *A-SHOP* algorithm. Thus, the difference between the total time and the running time indicates the sum of the communication time needed by *IMPACT* to access the remote information sources and of the time needed by the information sources to compute the answers to the queries.

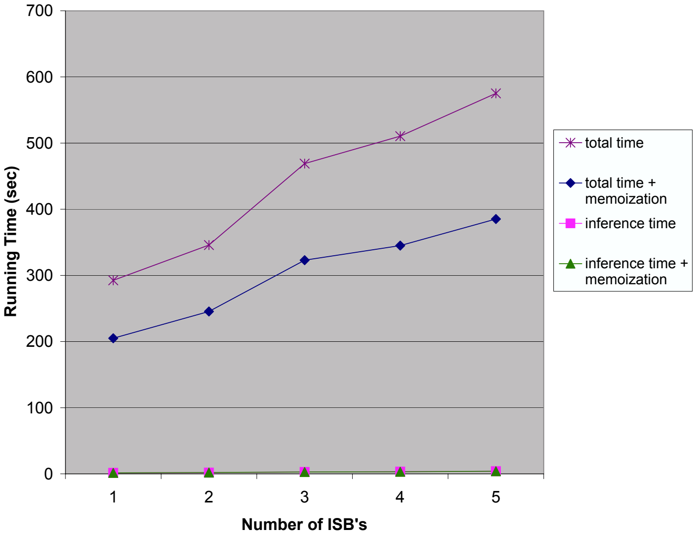

Figure 5: Results of the experiments.

Figure 5 shows the results of the experiments. Not surprisingly the inferencing times with and without memoization are almost identical. More interesting is the fact that the inferencing time is only a fraction of the overall running time. In addition, the use of the memoization mechanism results in a decrease in the running time of more than 30%.

7. RELATED WORK

Most AI planning systems are unable to evaluate numeric conditions at all. A few can evaluate numeric conditions using attached procedures (e.g., *SIPE* [23], *O-Plan* [3], *TLPlan* [2] and *SHOP* [16]), but the lack of a formal semantics for these attached procedures makes it more difficult to guarantee soundness and completeness. Integer Programming (IP) models appear to have excellent potential as a uniform formalism for reasoning about complex numeric and symbolic constraints during planning, and some work is already being done on the use of IP for reasoning about resources [14, 12, 24]. However, that work is still work in progress, and a number of fundamental problems still remain to be solved.

Approaches for planning with external information sources typically have in common that the information extracted from the external information sources is introduced in the planning system through built-in predicates [9, 11, 13, 10]. For example, a modified version of *UCPOP* uses *information gathering goals* to extract information from the external information sources [13]. The information gathering goals are used as preconditions of the operators. The primary difficulty with this approach is that since it is not clear what the semantic of the built-in predicates is, this makes it difficult to guarantee soundness and completeness.

Distributed problem-solving (eg. [4]) has been the focus of research for many years. With the advances in agent research [25], attention has been driven towards the coordination of the decision making process between multiple agents. However, much work is still needed in developing well-founded reasoning and negotiating techniques, in particular in environments in which the agent must constantly be on the lookout for changes (see [5] for a recent survey). An interesting approach is the *RETSINA* project [18, 19]. In *RETSINA* each agent can do its own planning, as each agent is equipped with a special planning component in its internal architecture. In contrast to this, we have chosen that one special planning agent, `shop`, does the planning upon request from other agents.

8. CONCLUSION

The original motivation of our work was to make *HTN* planning available in a multi-agent environment. This is beneficial for both, planners (they gain access to distributed and heterogenous information sources for free and can ship various tasks to other agents) as well as agent systems (which usually do not have available planning components that are highly sophisticated and efficient).

After developing the theory and implementing it, we ran experiments on a simplified version of the NEO domain, where data needed for the planning process is distributed and highly heterogenous. In such a situation, data changes dynamically, eg. weather conditions or available resources. Thus the available data can not be stored locally, because of the sheer amount and the dynamic changes in the database.

Our experiments revealed clearly that most of the time is spent on communication with the information sources and therefore network time. Thus improving the actual planning algorithm (as done by most planners that assume all info is there locally) does not pay off: the amount gained is orders of magnitude less than the overall time. We really need caching mechanisms, to avoid computing the same results over and over again. In the extreme case, when caching is just storing everything locally, we would end up with our original local planner. This is not feasible because of the amount of data involved and the fact that it changes dynamically. The other extreme is not to do any caching at all. Our memoization technique seems to be a good compromise between these two extremes. The decrease in time we are getting depends on the overall network time spent to access the information sources: the higher this network time, the higher is the gain obtained by our memoization technique. Consequently our experiments showed an overall gain ranging from 20%-40%.

9. REFERENCES

[1] F. Bacchus. The AIPS'00 Planning Competition. *AI Magazine*, 22(3), 2001.

[2] F. Bacchus and F. Kabanza. Using Temporal Logics to Express Search Control Knowledge for Planning. *Artificial Intelligence*, 116(1-2):123–191, 2000.

[3] K. Currie and A. Tate. O-plan: the open planning architecture. *Artificial Intelligence*, 52(1), 1991.

[4] R. Davis and R. Smith. Negotiation as a Metaphor for Distributed Problem Solving. *Artificial Intelligence*, 20(1), 1983.

[5] M. E. desJardins, E. H. Durfee, C. L. O. Jr., and M. J. Wolverton. A survey of research in distributed, continual planning. *AI Magazine*, 20(4), 1999.

[6] J. Dix, H. Munoz-Avila, D. Nau, and L. Zhang. IMPACTing SHOP: Putting an AI planner into a Multi-Agent Environment. *Annals of Mathematics and AI*, 2002. to appear.

[7] T. Eiter, V. Subrahmanian, and G. Pick. Heterogeneous Active Agents, I: Semantics. *Artificial Intelligence*, 108(1-2):179–255, 1999.

[8] T. Eiter, V. Subrahmanian, and T. J. Rogers. Heterogeneous Active Agents, III: Polynomially Implementable Agents. *Artificial Intelligence*, 117(1):107–167, 2000.

[9] O. Etzioni, D. Weld, D. Draper, N. Lesh, and M. Williamson. An approach to planning with incomplete information. In *Proceedings of KR-92*, 1992.

[10] M. Friedman and D. Weld. Efficiently executing information-gathering plans. In *Proceedings of IJCAI-97*, 1997.

[11] K. Golden, O. Etzioni, and D. Weld. Omnipotence without omniscience: efficient sensor management for planning. In *Proceedings of AAAI-94*, 1994.

[12] H. Kautz and J. P. Walser. State-space Planning by Integer Optimization. In *Proceedings of the 17th National Conference of the American Association for Artificial Intelligence*, pages 526–533, 1999.

[13] C. Knoblock. Building a planner for information gathering: a report from the trenches. In *Proceedings of AIPS-96*, 1996.

[14] J. Köhler. Planning under Resource Constraints. In *Proceedings of the 13th European Conference on Artificial Intelligence*, pages 489–493, 1998.

[15] H. Munoz-Avila, D. Aha, D. Nau, R. Weber, L. Breslow, and F. Yaman. Sin: Integrating case-based reasoning with task decomposition. In *Proceedings of IJCAI-01*, 2001.

[16] D. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila. Shop: Simple hierarchical ordered planner. In *Proceedings of IJCAI-99*, 1999.

[17] J. C. of Staff. Joint tactics, techniques and procedures for noncombatant evacuation operations. Technical Report Joint Report 3-07.51, Department of Defense, Washington DC, 1994.

[18] M. Paolucci, D. Kalp, A. Pannu, O. Shehory, and K. Sycara. A planning component for retsina agents. In M. Wooldridge and Y. Lesperance, editors, *Intelligent Agents VI*, 2000.

[19] M. Paolucci, O. Shehory, and K. Sycara. Interleaving planning and execution in a multiagent teamplanning environment. In *CMU-RI-TR-00-01*, 2000.

[20] L. K. S. Noncombatant evacuation operations: Plan now or pay later (technical report). In *Newport, RI: Naval War College*, 1992.

[21] A. Siegel. *Eastern Exit: The noncombatant evacuation operation (NEO) from Mogadishu, Somalia, in January 1991 (TR CRM 91-221)*. Arlington, VA: Center for Naval Analyses, 1991.

[22] V. Subrahmanian, P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Özcan, and R. Ross. *Heterogenous Active Agents*. MIT Press, 2000.

[23] D. Wilkins. *Practical planning - extending the classical AI planning paradigm*. Morgan Kaufmann, 1988.

[24] S. A. Wolfman and D. S. Weld. The LPSAT Engine and its Application to Resource Planning. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 310–317, 1999.

[25] M. Wooldridge and N. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Reviews*, 10(2), 1995.