IMPACTing SHOP: Putting an AI Planner into a Multi-Agent Environment

Jürgen Dix [a] Héctor Muñoz-Avila [b] Dana S. Nau [c] Lingling Zhang [d]

[a] *The University of Manchester, Oxford Road, Manchester, M13 9PL, UK*
E-mail: dix@cs.man.ac.uk

[b] *University of Maryland, College Park, MD 20742, USA,*
E-mail: munoz@cs.umd.edu

[c] *University of Maryland, College Park, MD 20742, USA,*
E-mail: nau@cs.umd.edu

[d] *University of Maryland, College Park, MD 20742, USA,*
E-mail: lingz@cs.umd.edu

In this paper we describe a formalism for integrating the SHOP HTN planning system with the IMPACT multi-agent environment. We define the A-SHOP algorithm, an agentized adaptation of the SHOP planning algorithm that takes advantage of IMPACT's capabilities for interacting with external agents, performing mixed symbolic/numeric computations, and making queries to distributed, heterogeneous information sources (such as arbitrary legacy and/or specialized data structures or external databases). We show that A-SHOP is both sound and complete if certain conditions are met.

Keywords: multi-agent systems, HTN planning, heterogenous/distributed data,

AMS Subject classification: Primary 68T20; Secondary 68T30

1. Introduction

In order to apply AI planning systems to complex real-world planning problems, here are some of the challenges that must be addressed:

1. *The need for the planning system to interact with external information sources* [CHWE95]. This problem tends to be complicated by the fact that the information sources are frequently heterogeneous and not necessarily centralized. For example, in an information integration project developed for the

US Army, the information sources included a US Army route planner over free terrain [BS94], a variety of US Army logistics data including specialized Oracle and nested multi-record TAADS data [SRM98], a variety of US Army simulation data from a massive program called JANUS, Training and Instrumentation Command, a face recognition program, and so forth.

2. *The need to perform mixed symbolic/numeric reasoning.* For example, [NSE98] describes the need to reason about a variety of numeric and symbolic conditions in order to do manufacturing planning and to plan declarer play in the game of bridge.

3. *The need to coordinate multiple agents.* For example, in planning the movement of a cargo container from its point of origin to its ultimate destination, a number of agents might participate in the control of the container: agents that load ships, higher level managers that react to unusual incidents, and so forth. The container agent would need to take these into account in plan development.[1]

Although a variety of approaches have been proposed for several of these challenges, none of them has yet been completely solved—and no current theory of planning addresses all of these challenges simultaneously.

The formalism described in this paper is intended to address all of the above challenges simultaneously, by integrating the SHOP planning system with the IMPACT multi-agent environment. While SHOP [NCLMA99] is a very efficient stand-alone HTN planner, the multi-agent platform IMPACT [SBD+00,ESP99] provides facilities for interacting with heterogeneous, distributed information sources (including arbitrary legacy and/or specialized data structures or external databases), combining symbolic and numerical information, and coordinating multiple agents. In this paper, we define A-SHOP (an agentized version of SHOP), and show how to realize it as an IMPACT agent (see Figure 2 in Section 3). This makes it possible for other IMPACT agents to communicate with A-SHOP and let it solve their own planning problems.

Although we have developed our formalism only for SHOP, we believe that a similar approach could be used to integrate other AI planners into IMPACT as well. The main new results in this paper are as follows:

[1] In this paper, we only consider the case where the other agents are information sources (rather than other planning agents). We intend to address the coordination of multiple planning agents in our future work.

- the definition of the A-SHOP planning algorithm, an agentized version of SHOP that runs in the IMPACT environment;
- the formulation of the conditions needed for A-SHOP to be sound and complete;
- proofs that A-SHOP is sound and complete if those conditions are met.

In addition, we are working on an implementation of our formalism, for an application domain involving *military logistics planning* (work currently in progress). This is an example of a domain where the SHOP-IMPACT framework can be very useful: information about the different assets is not centralized; and the information sources are heterogeneous, comprising different database management systems (DBMS).

This paper is organized as follows. In Section 2 we review *HTN planning*, the planning approach that SHOP is based on. In Section 3 we briefly introduce IMPACT, a platform for agents collaborating together. Section 4 contains the definition of A-SHOP (Figure 2 illustrates how to turn A-SHOP into a planning agent shop within IMPACT). Section 5 contains our main theorems, which establish the soundness and completeness of A-SHOP with respect to certain conditions on the code calls—conditions that were already well studied in IMPACT. Finally, we discuss related work in Section 7 and conclude with Section 8. The appendix contains detailed definitions of those concepts that could not be covered in the main part of the paper.

2. HTN Planning in SHOP

HTN planning [Sac77,Tat77,Wil88,CT91] is an AI planning methodology that creates plans by *task decomposition*. This is a process in which the planning system decomposes tasks into smaller and smaller subtasks, until primitive tasks are found that can be performed directly.

SHOP (see `<http://www.cs.umd.edu/projects/shop/>` and [NCLMA99, NMAC+01]) is a domain-independent Hierarchical Task Network (HTN) planning algorithm. However, one difference between SHOP and most other HTN planning algorithms is that SHOP plans for tasks in the same order that they will later be executed. Planning for tasks in the order that those tasks will be performed makes it possible to know the current state of the world at each step in the planning process, which makes it possible for SHOP's precondition-evaluation

mechanism to incorporate significant inferencing and reasoning power, including the ability to call external programs to reason about preconditions. This makes SHOP ideal as a basis for integrating planning with external information sources, such as the IMPACT framework as described in this paper.

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. More than one method may be applicable to the same task, in which case there will be more than one possible way to decompose that task.

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 1 shows two methods for the task of traveling from one location to another: *travelling by air*, and *travelling by taxi*. Travelling by air involves the subtasks of purchasing a plane ticket, travelling to the local airport, flying to an airport close to our destination, and travelling from there to our destination. Travelling by taxi involves the subtasks of *calling a taxi*, *riding in it to the final destination*, and *paying the driver*.

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. For example, in Figure 1, the *travel-by-air* method is only applicable for long distances, and the *travel-by-taxi* method is only applicable for short distances.

Now, consider the task of travelling from the University of Maryland to MIT. Since this is a long distance, the *travel-by-taxi* method is not applicable, so we must choose the *travel-by-air* method. As shown in Figure 1, this decomposes the task into the following subtasks: *(1)* purchase a ticket from Baltimore-Washington International (BWI) airport to Logan airport, *(2)* travel from the University of Maryland to BWI, *(3)* fly from BWI airport to Logan airport, and *(4)* travel from Logan airport to MIT. For the subtasks of travelling from the University of Maryland to BWI and traveling from Logan to MIT, we can use

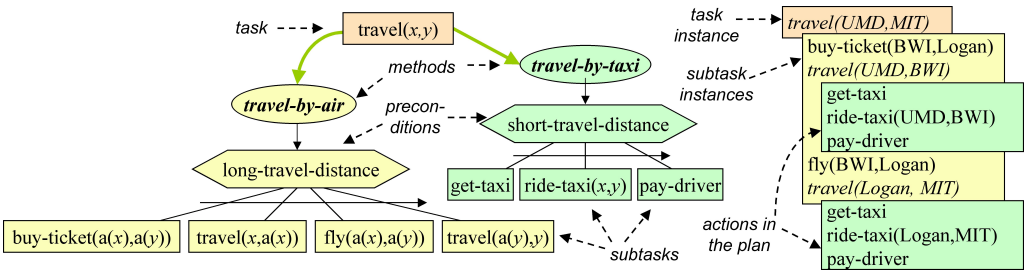

Figure 1. Two methods for traveling, and a plan generated from those methods.

the *travel-by-taxi* method to produce additional subtasks as shown in Figure 1.

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, we might want to make sure we will arrive at the airport in time to catch the plane. To make the example in Figure 1 more realistic, the information needed to enforce this constraint could be specified as part of SHOP's methods and operators.
- In the example in Figure 1, 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. IMPACT

The IMPACT project (see [ESP99,SBD+00] 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. The following points are of special interest for our integration of SHOP with IMPACT:

- IMPACT's methodology for transforming arbitrary software (legacy code) into an *agent* has been developed.
- Each agent has certain actions available. Agents act in their environment according to their *agent program* and a well defined *semantics*.
- IMPACT Agents are built on top of arbitrary software code (*Legacy Data*).
- Each agent continually undergoes the following cycle:

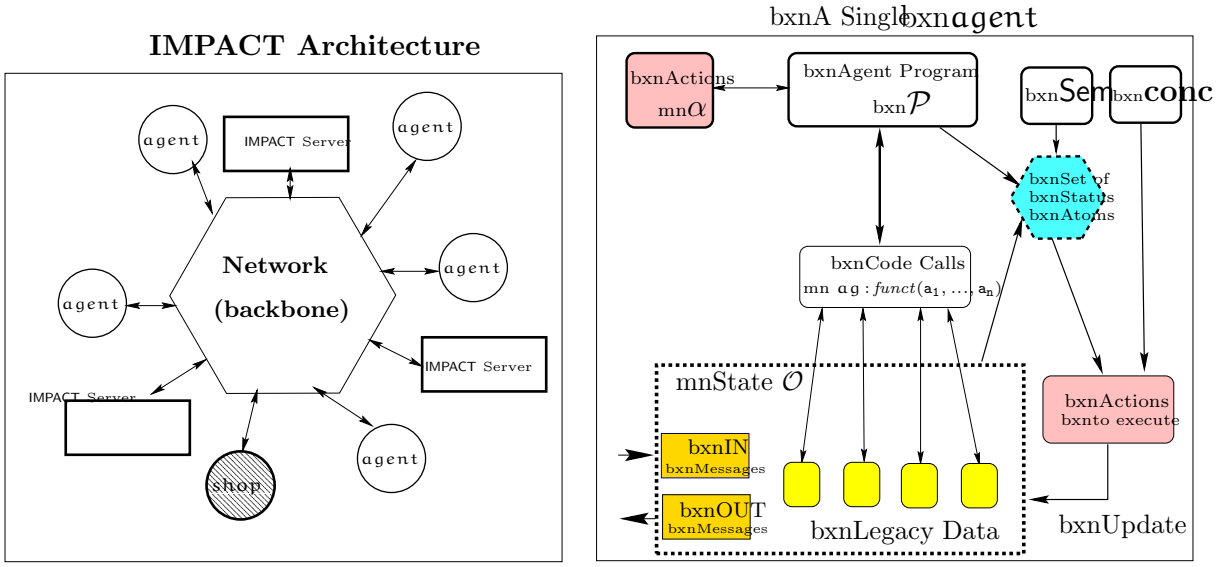

Figure 2. (1) SHOP as a planning agent in IMPACT. (2) An Agent in IMPACT.

(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.

A complete description of all these notions is out of scope of this paper, but the appendix contains some formal definitions. In the following, we concentrate only on those notions that are essential to understand the integration of SHOP in IMPACT.

Before explaining an agent in 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 (like 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 shop with information about relevant legacy data.

One of the main features of IMPACT is to provide a method (see [SBD+00]) 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 `statistics` agent is needed. This agent keeps track of distances between two given points and the authorized range or capacity of certain vehicules. These information can be stored in several databases. Another example is the `supplier` agent. It determines through its databases which vehicles are accessible at a given location.

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 two examples just mentioned, the state of the `statistics` agent consists of all tuples stored in the databases it handles. The state of the `supplier` agent is the set of all tuples describing which vehicles are accessible at a given location.

We also assume that agents can send and receive messages. There is therefore a special datastructure, the *message box*, part of each agent. This message box is just one of those types. Thus a state change occurs already when a message is received. A state, $\mathcal{O}$, can be seen as the union of the states of all agents in the environment.

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 1 (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 $\mathtt{d_1}, \ldots, \mathtt{d_n}$ are objects or variables such that each $\mathtt{d_i}$ respects the type requirements of the i'th argument of f. Then, $\mathcal{S}{:}f(\mathtt{d_1}, \ldots, \mathtt{d_n})$ is a *code call*. A code call is *ground* if all the $\mathtt{d_i}$'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 two agents introduced above, `statistics` may be able to execute the code calls `statistics` : *distance*(`locFrom`, `locTo`), `statistics` : *authorRange*(`CargoPL`), and `statistics` : *authorCapacity*(`CargoPL`). The `supplier` agent may execute the following code call: `supplier` : *cargoPlane*(`locFrom`).

What we really need to know is if the result of evaluating such code calls is contained in a certain set or not (for a precise description of the results of the code calls just mentioned we refer to Section C in the Appendix). 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 2 (Code Call Atoms (in(X, cc))). If `cc` is a code call, and `X` is either a variable symbol, or an object of the output type of `cc`, then **in**(`X`, `cc`) and **not_in**(`X`, `cc`) are *code call atoms*. **not_in**(`X`, `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 **in**(`X`, `cc`) succeeds if `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 code call atom **in**($f22$, `supplier` : *cargoPlane*(`collegepark`)) tells us that the particular plane "$f22$" is available as a cargo plane in College Park.

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.

The following definition expresses this intuition.

Definition 3 (Code Call Conditions (ccc)). A *code call condition* χ is defined as follows:

1. Every code call atom is a code call condition.
2. If `s`, `t` are either variables or objects, then `s` = `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.

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:

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

where $\alpha, \beta_1, \ldots \beta_n$ are *actions* (the agent can execute), $\mathsf{Op}_1, \ldots, \mathsf{Op}_n$ 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.

If we compare IMPACT actions with SHOP's methods, we easily notice that IMPACT actions correspond to fully instantiated methods, i.e. no subtasks.

Figure 2 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 concurrrency notion to finally determine which actions can be concurrently executed. The agent then executes these actions and changes its state.

The IMPACT framework has been extended to incorporate *time* ([DKS01]), *probabilities* ([DNS00]), *beliefs* ([DSP00]), and also mechanisms to merge many code calls together so that they can be more efficiently executed ([DOS00]).

4. A-SHOP: The IMPACT Version of SHOP

To plan with external information sources, A-SHOP's domain representation includes explicit calls to the IMPACT agents to enquire about their current states and to modify their states as we will explain now.

4.1. A-SHOP's Domain Definitions

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 below.

Definition 4 (Agentized Method: (AgentMeth $h\,\chi\,\mathbf{t}$)). An *agentized method* is an expression of the form **(AgentMeth** $h\,\chi\,\mathbf{t}$**)** where h (the method's *head*) is a compound task, χ (the method's *preconditions*) is a code call condition and $\mathbf{t}$ 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 5 (Agentized Operator: (AgentOp $h\,\chi_{add}\,\chi_{del}$)). An *agentized operator* is an expression of the form **(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.

As an example, Figure 3 shows a method for our application to military logistics planning. The method indicates how to transport a cargo that has a

certain weight between two locations. The method calls the statistics agent three times, in order to evaluate the *distance* between two geographic locations, the *authorized range* of a certain aircraft type (the authorized range is lower than the real distance that the aircraft can fly), and the *authorized capacity* of an aircraft (the amount of weight the aircraft can carry). The method calls the *supplier* agent to evaluate the cargo planes that are available at a location.

Head:
AirTransport(LocFrom, LocTo, Cargo, CargoWeight)

Preconditions:
in(CargoPL, supplier : *cargoPlane*(locFrom))&
in(Dist, statistics : *distance*(locFrom, locTo))&
in(DCargoPL, statistics : *authorRange*(CargoPL))&
Dist $\leq$ DCargoPL&
in(CCargoPL, statistics : *authorCapacity*(CargoPL))&
CargoWeight $\leq$ CCargoPL&

Subtasks:
load(Cargo, locFrom)
fly(Cargo, locFrom, LocTo)
unload(Cargo, locTo)

Figure 3. Agentized method for a military logistics problem.

The primary difference between the definition of an agentized operator and the definition of an operator in SHOP is that in an agentized operator, the elements of the add list and delete list are lists of code call rather than atoms. As described in the next section this has several consequences for how A-SHOP reasons about its current state of the world.

4.2. The A-SHOP Algorithm

The second step of the integration is to modify SHOP's HTN planning algorithm to use IMPACT. The modified algorithm, which we call the A-SHOP algorithm, is shown in Figure 4. Given a list **t** of tasks to achieve, and a collection $\mathcal{D}$ of agentized methods and operators, A-SHOP looks at the first task in the list. If that task is primitive (Step 3), then A-SHOP looks for a *simple plan* con-

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** $concatenate(q, \text{A-SHOP}(R, \mathcal{D}))$
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 μ the most general
   unifier of h and t and substitution θ s.t.
   $\chi\mu\theta$ is ground and holds in IMPACT's state $\mathcal{O}$.
8. **return** *A-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
    $op =$ **(AgentOp** $h\,\chi_{add}\,\chi_{del}$**)** with ν the most
    general unifier of h and t s.t. h is ground
12. `monitoring` : $apply(op\,\nu)$
13. **return** $op\,\nu$

end *A-SHOP*

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

sisting of a single operator (Step 4), and then calls itself recursively to examine the remaining tasks (Step 5), the function *concatenate* simply concatenates the plan q with the next plan that is obtained by the recursive call). If the task is compound and reducible (step 6), then A-SHOP applies an agentized method to produce subtasks (steps 7-8).

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 $cc = \mathcal{S} : f(\mathtt{d}_1, \ldots, \mathtt{d}_n)$ 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 everytime 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.

Like the original SHOP algorithm [NCLMA99], A-SHOP performs *ordered* task decomposition. This means that the task list $\mathbf{t}$ is totally ordered, the tasks need to be achieved in that same order, and SHOP always plans for the tasks in that same order. Like SHOP, A-SHOP needs to create partial plans, reason about the intermediate states of those partial plans, those partial plans might do, and backtrack if necessary to construct other partial plans. In SHOP, all of this is done inside the planning algorithm. However A-SHOP cannot do all of it internally because of the necessity of communicating with other agents to get information about the current state. In general, A-SHOP's preconditions will contain code calls to other agents, and the operators in its plans will contain code calls to other agents. Since A-SHOP may need to backtrack, it needs to reason about what the consequences will be of those code calls without actually telling the other agents to execute those code calls. To accomplish this, A-SHOP simulates calling the other agents by sending those code calls to a monitoring agent, `monitoring` (Steps 7 and 12).

This is the key point of the integration of SHOP in the IMPACT multi-agent environment: AI planners traditionally evaluate preconditions in a state, assumed to be internal to the planner. This assumption may be infeasible in many real-world situations. In A-SHOP, instead, the preconditions are evaluated externally

by IMPACT agents.

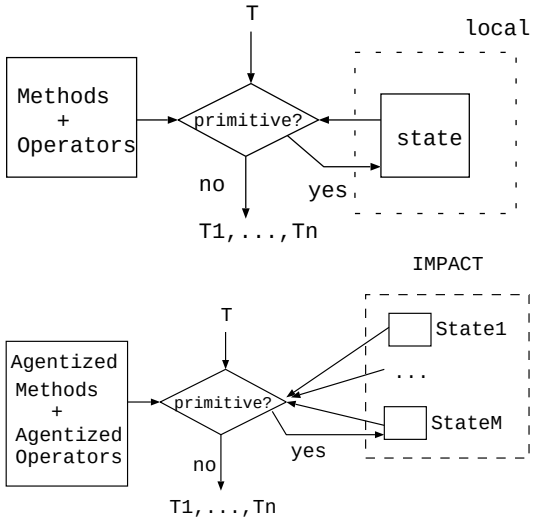

Figure 5. Data flow in (a) SHOP and (b) A-SHOP.

4.3. Realizing shop

The third step is to create an IMPACT agent, called shop, that acts as a planning agent and performs the A-SHOP algorithm. To accomplish this, we can use the general-purpose agentizing algorithm described in [SBD+00]. This enables A-SHOP to communicate with other IMPACT agents and vice-versa. Basically, the agentizing algorithm outputs a protocol that presents the procedure calls of the software in a standardized format that allows other agents to communicate with it. For the particular situation of a planning system, the protocol includes a call to a procedure that receives as input a problem description and outputs a solution plan (please refer to [SBD+00] for a detailed discussion of the algorithm).

5. Soundness and Completeness

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 are statements in a logical language referring to arbitrary software functions. As an example, we consider a software package `math` which provides several functions to deal with integers. The code call $\texttt{math} : \mathit{geq}(\texttt{X})$ enumerates all integers greater or equal to `X`. This cc is not executable, because it is not ground. Only if the variable `X` is assigned a certain value, the code call can be executed. A syntactical condition to ensure this is safeness (see appendix). But safeness is not enough! Consider the code call

$$\begin{aligned}&\mathbf{in}(\texttt{X}, \texttt{math} : \mathit{geq}(\texttt{25}))\,\&\\&\mathbf{in}(\texttt{Y}, \texttt{math} : \mathit{square}(\texttt{X}))\,\&\,\texttt{Y} \leq 2000,\end{aligned}$$

which constitutes all numbers that are less than 2000 and that are squares of an integer greater than or equal to 25.

Clearly, over the integers there are only finitely many ground substitutions that cause this code call condition to be true. Furthermore, this code call condition is safe. However, its evaluation may never terminate. The reason for this is that safety requires that we first compute the set of all integers that are greater than 25, leading to an infinite computation.

Thus in general, we must impose some restrictions on code call conditions to ensure that they are finitely evaluable. This is precisely what the condition of strongly safeness does for the code-call conditions. Intuitively, by requiring that the code call condition is safe, we are ensuring that it is executable and by requiring that it is strongly safe, we are ensuring that it will only return finitely many answers.

Note that the problem of deciding whether an arbitrary code call execution terminates is undecidable (and so is the problem of deciding whether a code call condition χ holds in $\mathcal{O}$). Therefore we need some input of the agent designer (or of the person who is responsible for the legacy code the agent is built upon). The information needed is stored in a *finiteness table* (see Definition 13). This information is used in the *purely syntactic* notion of strong safeness. It is a

compile-time check, an extension of the well-known (syntactic) safety condition in databases.

Lemma 6 (Evaluating Agentized Operators).
Let $\mathcal{O}$ be a state, **(AgentMeth** $h\,\chi\,\mathbf{t}$**)** an agentized method 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.

Proof: This follows immediately from results in Chapter 12 of [SBD$^+$00]. ∎

Theorem 1 (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.

Proof: See Appendix E. ∎

6. Implementation

A version of IMPACT is running on a Windows platform. This version has been built primarily in JAVA. The implementation of IMPACT uses a pre-existing software package developed at the University of Maryland called *WebHermes* [Ada97] which supports execution of code call conditions over a wide variety of data structures and software packages. These currently include (or have included in the past), relational database management systems (Oracle, Ingres, Dbase, Paradox), an object oriented system (ObjectStore), a multimedia system called MACS [BMS95], a video information system called AVIS [ACC$^+$96], a geographic data structure called a PR-quadtree, arbitrary flat files (as long as their schemas are specified), a US Army route planner over free terrain [BS94], a variety of US Army logistics data including specialized Oracle and nested multirecord TAADS data [SRM98], a variety of US Army simulation data from a massive program

called JANUS deployed by the Simulation, Training and Instrumentation Command, a face recognition program, and so on.

The complete version of SHOP is built in LISP and includes the abilities to do Horn-clause inferencing and to make calls to the LISP evaluator. The former one is used to infer conditions from the current state and the latter one is used to add expressiveness during planning. For example, SHOP can compute numerical expressions. SHOP can be downloaded from `<http://www.cs.umd.edu/projects/shop/>`. To facilitate the integration of SHOP in IMPACT, we re-implemented SHOP in java. The java version of SHOP includes neither the Horn clause evaluator nor the calls to the LISP evaluator. However, such things could easily be added through the use of the IMPACT framework, without needing any modifications to the current JAVA implementation of SHOP. In particular, we could take an arbitrary theorem prover and *agentize* it using IMPACT methods. The agent could then be called using an appropriate code-call condition ([SBD+00] describes a step by step process to agentize a program and incorporate it as an agent into IMPACT). The same can be done for evaluations. In particular, a mathematical agent, `math`, is currently available in IMPACT to evaluate some numerical expressions.

A difficulty that we found in implementing the A-SHOP algorithm is that rules defining agents in IMPACT are not **recursive over time**: they do not formalize statements of the form *If some code calls are executed at time t then some other facts hold true at time $t+1$*. In fact, the basic framework of IMPACT does not even express statements of the form *If something is true at time t then something else should be true at time $t+1$*. Such an extension is described in [DKS01].

In our version of IMPACT, the status set is computed for a particular point in time t. Thus while agent rules are recursive, they always determine a status set for a single time point. This status set together with a particular evaluation strategy determines the set of actions to be executed. Agent rules do not describe the effects of these actions.

This made it necessary to define four agents to implement the A-SHOP algorithm:

- `Ashop`. This is the agent that all other agents access to generate a plan. It receives a message with a domain and a list of tasks and returns a plan achieving all tasks if such a plan exists or returns failure if no such plan exists.

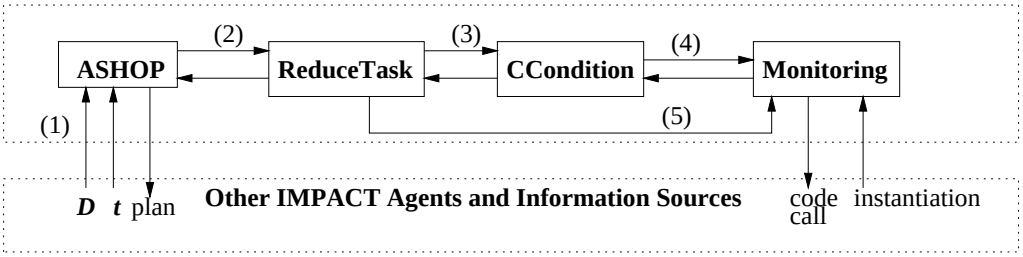

Figure 6. Interactions between the agents implementing the A-SHOP algorithm.

- ReduceTask. Selects the next task to be reduced and performs a single reduction step by selecting either a method or an operator.
- CCCondition. Evaluates a code call condition (i.e., a list of code calls).
- Monitoring. Maintains a list of actions to be executed. If a code call $\mathcal{S}:f(\mathtt{d}_1,\ldots,\mathtt{d}_n)$ is to be evaluated, it looks at its action list to determine if the code call can be answered as a result of its actions. If not it queries the code call to the information source $\mathcal{S}$.

Figure 6 describes how these four agents interact. First, a message requesting a plan is (1) received by the Ashop agent. This message contains a domain, $\mathcal{D}$, and a list of tasks, $\mathbf{t}$, to be achieved. Ashop initializes some variables and (2) passes them to ReduceTask together with $\mathbf{t}$ and $\mathcal{D}$. If the first task, t, in $\mathbf{t}$ is compound, it selects a method whose head matches t and (3) passes its preconditions to CCCondition. CCCondition iterates over the preconditions (4) passing one by one to Monitoring for evaluation. If t is primitive, it selects an operator whose head matches t and (5) passes the instantiated operator to Monitoring to update its action list.

Steps (1) and (2) are executed once in every planning episode. Step (3) is executed for each selected method. Step (4) is executed for every code call in the method's preconditions and Step (5) is executed for each selected operator.

We plan to conduct experiments in a simplified version of planning for Noncombatant evacuation operations (*NEO*'s). *NEO*'s are 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, case studies [Sie91], and more general analyses (e.g., [S.92]).

In our experiments we use a knowledge base that we developed for *NEO* planning. Its plans concern 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). Once the troops arrive to the *NEO* site, evacuees are re-located in a safe haven. Planning involves selecting possible predefined routes, consisting of four segments each. The planner must also choose means of transportation for each segment. The knowledge base included six agentized operators and 22 agentized methods. There are five information sources, one for each location, which include the assembly point and the *NEO* site. The other locations are: the ISB (intermediate staging base, where the troops are organized previous to accessing the *NEO* site), the FAARP (where logistics equipment is grouped) and the Safe Haven (where the evacuees are re-located once the operation ends). These information sources contain information about the corresponding locations, including: (1) which assets are available, (2) what are the meterological conditions and (3) information about enemy presence in that area.

Different agentized methods are triggered depending on the particular information maintained in an specific location. For example, selecting an agentized method taking the air transport to re-locate the evacuees between the *NEO* Site and the Safe Haven requires that (1) the *NEO* Site and the Safe Haven have airports, (2) that sufficinet airplanes are available at the *NEO* Site and (3) that the methereological conditions in the *NEO* Site and the Safe Haven are appropriate. Each of these requirements is expressed in several code calls, in which each of the information sources is queried to establish if the requirements are met.

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 [Wil88], O-Plan [CT91], TLplan [BK00] and SHOP [NCLMA99]), 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 [Köh98,KW99,WW99]. 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 [EWD+92,GEW94, Kno96,FW97]. For example, a modified version of UCPOP uses *information gathering goals* to extract information from the external information sources [Kno96]. 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. [DS83]) has been the focus of research for many years. With the advances in agent research [WJ95], 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 [dDJW99] for a recent survey). An interesting approach is the RETSINA project [PKP+00,PSS00]. 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.

PLACA (PLAnning Communicating Agents) is an agent-oriented language [Tho93,Tho94]. PLACA adds two mechanisms to the agent's state : (consistent) list of intentions and a list of (consistent) plans. Intentions are adopted via commands expressed in the PLACA language. Plans are created by an external plan generator to meet these intentions. In A-SHOP the intensions are indicated in the tasks to be achieved and the A-SHOP planning algorithm perform a task decomposition process that results in the plans achieving those tasks (i.e., meeting the intentions).

AgentSpeak(L) [Rao96] is a Believe-Desire-Intention (BDI) language and, like

PLACA, capable of expressing beliefs, commitments and communications between agents. However, AgentSpeak(L) provides operational and proof-theoretic semantics and does not provide any planning features. In this paper we demonstrated A-SHOP's operational semantics by showing that it is sound and complete.

3APL ([HBdHM99]) is a general agent programming language the operational semantics of which is based on transition systems. It inherits quite a lot of regular programming constructs from imperative programming, such as recursive procedures but is built upon clear formal semantics. As a very general agent programming language, it is much more general than IMPACT's agent programs. However, there is no specialized planning component available. Certainly, planning algorithms might be implemented in this language, but there is no particular support for that. It has been shown that various languages such as AgentSpeak(L) or ConGolog can be easily embedded in 3APL.

8. Conclusion

We have developed A-SHOP, a modified version of the SHOP planning algorithm that takes advantage of the capabilities provided by the IMPACT multi-agent environment. A-SHOP can plan with heterogeneous, distributed information sources, combine symbolic and numerical information, and interact with multiple agents.

In the A-SHOP algorithm, SHOP's preconditions, add-lists and delete-lists are replaced with code call conditions. IMPACT's code call conditions provide both well-defined syntax and a well-defined semantics. This has enabled us to show that A-SHOP is sound and complete provided that the code calls are strongly safe.

Using the methodology outlined in [SBD+00], it is straightforward to turn A-SHOP into a planning agent `shop`.

A-SHOP, like most AI planners, normally constructs entire plans before beginning plan execution. In a multi-agent setting, it would be desirable to have the ability to interleave planning with plan execution. In the current A-SHOP algorithm, this could be accomplished by giving A-SHOP a set of methods and operators that tell it to produce a high-level plan $(p_1, \ldots, p_k)$ in which each p_i is itself a call to A-SHOP. However, since this approach is somewhat *ad hoc*, one of our topics for future work is to develop an extension to the A-SHOP formalism that explicitly interleaves planning with plan execution.

Note that unlike most AI planning algorithms, A-SHOP does not have any

information about its state stored locally. However, we could if needed simulate having a local state by simply defining an agent that manages the state and having all code call conditions refer to that agentized state. Intermediate approaches such as Knoblock's [Kno96], which updates the current state by gathering information from external sources, can also be subsumed in our integration: again we could have an specialized agent managing the partial state of the world.

A-SHOP's use of IMPACT's code call atoms allows us to address the challenges stated in the introduction:

- **Mixed symbolic/numeric reasoning.** Notice that the precondition of the method shown in Figure 3 supposes a combination symbolic and numeric reasoning. On one hand, this method is used as a means for decomposing the task

  $$\mathit{AirTransport}(\texttt{LocFrom}, \texttt{LocTo}, \texttt{Cargo}, \texttt{CargoWeight}),$$

  which is essentially a symbolic process. On the other hand, some of its preconditions are numerical comparisons (i.e., $\texttt{Dist} \leq \texttt{DCargoPL}$). This is a simple illustration of a greater potential: by decoupling the evaluation of preconditions from the planning process itself we are gaining flexibility. Specialized agents performing complex numerical calculations can be plugged in.
- **Distributed, heterogeneous information sources.** One important effect of integrating A-SHOP within IMPACT is that it allows A-SHOP to gather information from distributed, heterogeneous information sources without requiring knowledge about how and where these resources are located. For example, in the method shown in Figure 3, determining the statistics of a certain airplane might simply require access to a local database, but determining if any such airplanes are available in a certain location might require access to a remotely located spreadsheet.

  It is interesting to note that according to a recent study, handling resources separately from the planning process can improve the performance of planning systems [SK99]. We have not yet examined whether such an improvement occurs in A-SHOP (as opposed to SHOP), but we hope to be able to examine it in the future.
- **Coordination of multiple agents.** Every time A-SHOP does a code call, a request to contact an external agent is made. The IMPACT multi-agent environment coordinates this process. In principle, this could be used not only

to communicate between A-SHOP and the other agents, but also to coordinate multiple versions of A-SHOP itself. We have not yet implemented multiple copies of A-SHOP running concurrently, but we hope to do so in the near future.

Acknowledgements

The authors would like to thank two anonymous referees. Their careful reading of a draft of this paper greatly improved its presentation.

This work was supported in part by the following grants, contracts, and awards: Air Force Research Laboratory F306029910013 and F30602-00-2-0505, Army Research Laboratory DAAL0197K0135, and the University of Maryland General Research Board. Opinions expressed in this paper are those of authors and do not necessarily reflect opinion of the funders.

References

[ACC+96] S. Adali, K. S. Candan, S.-S. Chen, K. Erol, and V. S. Subrahmanian. Advanced Video Information Systems: Data Structures and Query Processing. *Multimedia Systems*, 4(4):172–186, 1996.

[Ada97] Adali, S., et al. Web hermes user manual, 1997. `http://www.cs.umd.edu/projects/hermes/UserManual/index.html`.

[BK00] F. Bacchus and F. Kabanza. Using Temporal Logics to Express Search Control Knowledge for Planning. *Artificial Intelligence*, 116(1-2):123–191, 2000.

[BMS95] A. Brink, S. Marcus, and V. S. Subrahmanian. Heterogeneous Multimedia Reasoning. *IEEE Computer*, 28(9):33–39, 1995.

[BS94] John Benton and V. S. Subrahmanian. Using Hybrid Knowledge Bases for Missile Siting Problems. In IEEE Computer Society, editor, *Proceedings of the Conference on Artificial Intelligence Applications*, pages 141–148, 1994.

[CHWE95] S. Chien, R. Hill, X. Wang, and T. Estlin. Why real-world planning is difficult: A tale of two applications. In *Proceedings of the 3rd Europ. Workshop on Planning (EWSP-95)*, 1995.

[CT91] K. Currie and A. Tate. O-plan: the open planning architecture. *Artificial Intelligence*, 52(1), 1991.

[dDJW99] M. E. desJardins, E. H. Durfee, C. L. Ortiz Jr., and M. J. Wolverton. A survey of research in distributed, continual planning. *AI Magazine*, 20(4), 1999.

[DKS01] Jürgen Dix, Sarit Kraus, and V.S. Subrahmanian. Temporal agent programs. *Artificial Intelligence*, 127(1):87–135, 2001.

[DNS00] Jürgen Dix, Mirco Nanni, and V.S. Subrahmanian. Probabilistic Agent Programs. *ACM Transactions of Computational Logic*, 1(2):207–245, 2000.

[DOS00] Jürgen Dix, Fatma Öczan, and V.S. Subrahmanian. Improving performance of heavily loaded agents. Technical Report CS-TR-4202, Dept. of CS, University of Maryland, College Park, MD 20752, November 2000.

[DS83] R. Davis and R. Smith. Negotiation as a Metaphor for Distributed Problem Solving. *Artificial Intelligence*, 20(1), 1983.

[DSP00] Jürgen Dix, V.S. Subrahmanian, and George Pick. Meta Agent Programs. *Journal of Logic Programming*, 46(1-2):1–60, 2000.

[ES99] T. Eiter and V. S. Subrahmanian. Heterogeneous Active Agents, II: Algorithms and Complexity. *Artificial Intelligence*, 108(1-2):257–307, 1999.

[ESP99] Thomas Eiter, V. S. Subrahmanian, and Georg Pick. Heterogeneous Active Agents, I: Semantics. *Artificial Intelligence*, 108(1-2):179–255, 1999.

[ESR00] T. Eiter, V.S. Subrahmanian, and T J. Rogers. Heterogeneous Active Agents, III: Polynomially Implementable Agents. *Artificial Intelligence*, 117(1):107–167, 2000.

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

[FW97] M. Friedman and D. Weld. Efficiently executing information-gathering plans. In *Proceedings of IJCAI-97*, 1997.

[GEW94] K. Golden, O. Etzioni, and D. Weld. Omnipotence without omniscience: efficient sensor management for planning. In *Proceedings of AAAI-94*, 1994.

[HBdHM99] Koen V. Hindriks, Frank S. De Boer, Wiebe Van der Hoek, and John-Jules Ch. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

[Kno96] C.A. Knoblock. Building a planner for information gathering: a report from the trenches. In *Proceedings of AIPS-96*, 1996.

[Köh98] J. Köhler. Planning under Resource Constraints. In *Proceedings of the 13th European Conference on Artificial Intelligence*, pages 489–493, 1998.

[KW99] Henry Kautz and Joachim 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.

[NCLMA99] D.S. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila. Shop: Simple hierarchical ordered planner. In *Proceedings of IJCAI-99*, 1999.

[NMAC+01] D.S. Nau, H. Muñoz-Avila, Y. Cao, A. Lotem, and S.Mitchell. Total-order planning with partially ordered subtasks. In *Proceedings of IJCAI-2001*, 2001.

[NSE98] Dana S. Nau, Stephen J. J. Smith, and Kutluhan Erol. Control Strategies in HTN Planning: Theory versus Practice. In *AAAI-98/IAAI-98 Proceedings*, pages 1127–1133, 1998.

[PKP+00] 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.

[PSS00] M. Paolucci, O. Shehory, and K. Sycara. Interleaving planning and execution in a multiagent teamplanning environment. In *CMU-RI-TR-00-01*, 2000.

[Rao96] A. Rao. BDI agents speak out in a logical computable language. In *In W. Van*

deVelde and J. W. Perram, editors, Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, LNAI 1038, pages 42–55. Springer, 1996.

[S.92] Lambert Kirk S. Noncombatant evacuation operations: Plan now or pay later (technical report). In *Newport, RI: Naval War College*, 1992.

[Sac77] E. Sacerdoti. *A Structure for Plans and Behavior*. American Elsevier Publishing, 1977.

[SBD+00] V.S. Subrahmanian, Piero Bonatti, Jürgen Dix, Thomas Eiter, Sarit Kraus, Fatma Özcan, and Robert Ross. *Heterogenous Active Agents*. MIT Press, 2000.

[Sie91] A.B. 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.

[SK99] B. Srivastava and S. Kambhampati. Scaling up planning by teasing out resource scheduling. In *ASU CSE TR 99-005. To appear in ECP-99.*, 1999.

[SRM98] J. Schafer, T. J. Rogers, and J.A. Marin. Networked Visualization of Heterogeneous US Army War Reserves Readiness Data. In S. Jajodia, T. Ozsu, and A. Dogac, editors, *Advances in Multimedia Information Systems, 4th International Workshop, MIS'98*, volume 1508 of *Lecture Notes in Computer Science*, pages 136–147, Istanbul, Turkey, September 1998. Springer-Verlag.

[Tat77] A. Tate. Generating Project Networks. In *Proc. IJCAI-77*, pages 888–893, 1977.

[Tho93] S. Thomas. *PLACA: An Agent-Oriented Programming Language*. PhD thesis, Stanford University, 1993.

[Tho94] S. Thomas. The placa agent programming language. In *Pre-Proc. of Workshop on Agent Theories, Architectures, and Languages (ATAL'94), Amsterdam, The Netherlands*, pages 307–319, 1994.

[Wil88] D.E. Wilkins. *Practical planning - extending the classical AI planning paradigm*. Morgan Kaufmann, 1988.

[WJ95] M. Wooldridge and N. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Reviews*, 10(2), 1995.

[WW99] Steven A. Wolfman and Daniel 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.

Appendix

A. IMPACT

The following definition specifies what a *solution* of a code call condition is. Intuitively, code call conditions are evaluated against an agent state—if the state of the agent changes, the solution to a code call condition may also undergo a change.

Definition 7 (Code Call Solution). Suppose χ is a code call condition involving the variables $\mathbf{X} =_{def} \{X_1, \ldots, X_n\}$, and suppose $\mathcal{S} =_{def} (\mathcal{T}_\mathcal{S}, \mathcal{F}_\mathcal{S}, \mathcal{C}_\mathcal{S})$ is some software code. A *solution* of χ w.r.t. $\mathcal{T}_\mathcal{S}$ in a state $\mathcal{O}_\mathcal{S}$ is a legal assignment of objects $o_1, \ldots, o_n$ to the variables $X_1, \ldots, X_n$, written as a compound equation $\mathbf{X} := \mathbf{o}$, such that the application of the assignment makes χ true in state $\mathcal{O}_\mathcal{S}$.

We denote by $\mathsf{Sol}(\chi)_{\mathcal{T}_\mathcal{S},\mathcal{O}_\mathcal{S}}$ (omitting subscripts $\mathcal{O}_\mathcal{S}$ and $\mathcal{T}_\mathcal{S}$ when clear from the context), the set of all solutions of the code call condition χ in state $\mathcal{O}_\mathcal{S}$, and by $\mathcal{O}\_\mathsf{Sol}(\chi)_{\mathcal{T}_\mathcal{S},\mathcal{O}_\mathcal{S}}$ (where subscripts are occasionally omitted) the set of all objects appearing in $\mathsf{Sol}(\chi)_{\mathcal{T}_\mathcal{S},\mathcal{O}_\mathcal{S}}$

Here is the notion of an action as used in IMPACT.

Definition 8 (Action; Action Atom). An *action* α consists of six components:

Name: A name, usually written $\alpha(\mathtt{X}_1, \ldots, \mathtt{X}_n)$, where the $\mathtt{X}_i$'s are root variables.
Schema: A schema, usually written as $(\tau_1, \ldots, \tau_n)$, of types. Intuitively, this says that the variable $\mathtt{X}_i$ must be of type τ_i, for all $1 \leq i \leq n$.
Action Code: This is a body of code that executes the action.
Pre: A code-call condition χ, called the *precondition* of the action, denoted by $Pre(\alpha)$ ($Pre(\alpha)$ must be *safe modulo the variables* $\mathtt{X}_1$,...,$\mathtt{X}_n$);
Add: a set $Add(\alpha)$ of code-call conditions;
Del: a set $Del(\alpha)$ of code-call conditions.

We close with the definition of *action atom*. An *action atom* is a formula $\alpha(t_1, \ldots, t_n)$, where t_i is a term, i.e., an object or a variable, of type τ_i, for all $i = 1, \ldots, n$.

What is the difference with this approach and the classical AI framework?

Item	**Classical AI**	**Our framework**
State $\mathcal{O}$	Set of logical atoms	Data structures
Prec	Logical formula	Code call condition
Add/Del	set of ground atoms	Code call condition
Impl.	Via add/delete lists	Via arbitrary code
Reas.	Via add/del lists	Via add/del lists

States in classical AI planning are *physically modified* by union-ing the current state with items in the add list, and then deleting items in the delete list.

In contrast, the add-list and the delete-list in our framework plays no role whatsoever in the physical implementation of the action. The action is implemented by its associated action code. The agent uses the preconditions, add list, and the delete list to reason about what is true/false in the new state.

Definition 9 (Status Atom/Status Set). If $\alpha(\vec{t})$ is an action, and $\mathsf{Op} \in \{\mathbf{P}, \mathbf{F}, \mathbf{W}, \mathbf{Do}, \mathbf{O}\}$, then $\mathsf{Op}\alpha(\vec{t})$ is called a *status atom.* If A is a status atom, then $A, \neg A$ are called *status literals.* A *status set* is a finite set of ground status atoms.

Intuitively, $\mathbf{P}\alpha$ means α is permitted, $\mathbf{F}\alpha$ means α is forbidden, $\mathbf{Do}\,\alpha$ means α is actually done, $\mathbf{O}\alpha$ means α is obliged, and $\mathbf{W}\alpha$ means that the obligation to perform α is waived.

Definition 10 (Agent Program). An agent program $\mathcal{P}$ is a finite set of rules of the form

$$A \leftarrow \chi \,\&\, L_1 \,\&\, \ldots \,\&\, L_n$$

where χ is a code call condition and $L_1, \ldots, L_n$ are status literals.

The semantics of agent programs are described in [ESP99,ES99,SBD$^+$00].

B. Termination of Code Calls

Note that the following definitions underlying strongly safeness are important for the general case, when an agent is built upon arbitrary code. For most applications, like our military logistics domain, the safety requirement is completely sufficient.

As already mentioned, strongly safeness is a condition to ensure that code calls are not only evaluable, but that they generate only finitely many answers (they eventually terminate) [ESR00]. Now given an arbitrary function, the problem of deciding whether its range is finite or not is well-known to be undecidable. Therefore we need help from the system designer. The key notion is the following.

Definition 11 (Binding Pattern). Suppose we consider a code call $\mathcal{S}:f(\mathtt{a_1}, \ldots, \mathtt{a_n})$ where each $\mathtt{a_i}$ is of type τ_i. A *binding pattern* for $\mathcal{S}:f(\mathtt{a_1}, \ldots, \mathtt{a_n})$ is an n-tuple $(bt_1, \ldots, bt_n)$ where each bt_i (called a *binding term*) is either:

1. A value of type τ_i, or

2. The expression $\flat$ denoting that this argument is bound to an unknown value.

We require that the agent developer must specify a *finiteness* predicate that may be defined via a *finiteness table* having two columns—the first column is the name of the code call, while the second column is a binding pattern for the function in question. Intuitively, suppose we have a row of the form

$$\langle \mathcal{S} : f(\mathtt{a_1}, \mathtt{a_2}, \mathtt{a_3}), (\flat, 5, \flat) \rangle$$

in the finiteness table. Then this row says that the answer returned by any code call of the form $\mathcal{S} : f(-, 5, -)$ is finite. In other words, as long as the second argument of this code call is 5, the answer returned is finite, irrespective of the values of the first and third arguments. Clearly, the same code call may occur many times in a finiteness table with different binding patterns.

Definition 12 (Ordering on Binding Patterns). We say a binding pattern $(bt_1, \ldots, bt_n)$ is *equally or less informative* than another binding pattern $(bt'_1, \ldots, bt'_n)$ *if, by definition,* for all $1 \leq i \leq n$, $bt_i \leq bt'_i$.

We will say $(bt_1, \ldots, bt_n)$ is *less informative* than $(bt'_1, \ldots, bt'_n)$ *if and only if* it is equally or less informative than $(bt'_1, \ldots, bt'_n)$ and $(bt'_1, \ldots, bt'_n)$ is not equally or less informative than $(bt_1, \ldots, bt_n)$. If $(bt'_1, \ldots, bt'_n)$ is less informative than $(bt_1, \ldots, bt_n)$, then we will say that $(bt_1, \ldots, bt_n)$ is *more informative* than $(bt'_1, \ldots, bt'_n)$.

Suppose now that the developer of an agent specifies a finiteness table FINTAB. The following definition specifies what it means for a specific code call atom to be considered finite w.r.t. FINTAB.

Definition 13 (Finiteness). Suppose FINTAB is a finite finiteness table , and $(bt_1, \ldots, bt_n)$ is a binding pattern associated with the code call $\mathcal{S} : f(\cdots)$. Then FINTAB is said to *entail the finiteness of* $\mathcal{S} : f(\mathtt{bt_1}, \ldots, \mathtt{bt_n})$ *if, by definition,* there exists an entry of the form $\langle \mathcal{S} : f(\ldots), (bt'_1, \ldots, bt'_n) \rangle$ in FINTAB such that $(bt_1, \ldots, bt_n)$ is more informative than $(bt'_1, \ldots, bt'_n)$.

Definition 14 (Strong Safety). A safe code call condition $\chi = \chi_1 \,\&\, \ldots \,\&\, \chi_n$ is *strongly safe* w.r.t. a list $\vec{X}$ of root variables *if, by definition,* there is a

permutation π witnessing the safety of χ modulo $\vec{X}$ such that for each $1 \leq i \leq n$, $\chi_{\pi(i)}$ is strongly safe modulo $\vec{X}$, where strong safety of $\chi_{\pi(i)}$ is defined as follows:

1. $\chi_{\pi(i)}$ *is a code call atom.*
   Here, let the code call of $\chi_{\pi(i)}$ be $\mathcal{S}:f(\mathtt{t_1},\ldots,\mathtt{t_n})$ and let the binding pattern $\mathcal{S}:f(\mathtt{bt_1},\ldots,\mathtt{bt_n})$ be defined as follows:
   (a) If t_i is a value, then $bt_i = t_i$.
   (b) Otherwise t_i must be a variable whose root occurs either in $\vec{X}$ or in $\chi_{\pi(j)}$ for some $j < i$. In this case, $bt_i = \flat$.

   Then, $\chi_{\pi(i)}$ is strongly safe *if, by definition,* FINTAB entails the finiteness of $\mathcal{S}:f(\mathtt{bt_1},\ldots,\mathtt{bt_n})$.
2. $\chi_{\pi(i)}$ *is* $\mathtt{s} \neq \mathtt{t}$.
   In this case, $\chi_{\pi(i)}$ is strongly safe *if, by definition,* each of $\mathtt{s}$ and $\mathtt{t}$ is either a constant or a variable whose root occurs either in $\vec{X}$ or in $\chi_{\pi(j)}$ for some $j < i$.
3. $\chi_{\pi(i)}$ *is* $\mathtt{s} < \mathtt{t}$ *or* $\mathtt{s} \leq \mathtt{t}$.
   In this case, $\chi_{\pi(i)}$ is strongly safe *if, by definition,* $\mathtt{t}$ is either a constant or a variable whose root occurs either in $\vec{X}$ or somewhere in $\chi_{\pi(j)}$ for some $j < i$.
4. $\chi_{\pi(i)}$ *is* $\mathtt{s} > \mathtt{t}$ *or* $\mathtt{s} \geq \mathtt{t}$.
   In this case, $\chi_{\pi(i)}$ is strongly safe *if, by definition,* $\mathtt{t} < \mathtt{s}$ or $\mathtt{t} \leq \mathtt{s}$, respectively, are strongly safe.

Algorithms to check strong safety are developed in [SBD+00].

C. Example of an Application Domain

Military logistics planning is an example of a domain where the SHOP-IMPACT framework can be very useful. In particular with respect to logistics planning for the US Armed Forces: first, information about the different assets is not centralized, second, the information sources are heterogeneous, comprising different database management systems (DBMS).

Figure 7 shows some of the code-calls for this application. The first three code-calls access the agent `statistics` and return the *distance* between two geographic locations, the *authorized range* of a certain aircraft type (the authorized range is lower than the real distance that the aircraft can fly), and the

authorized capability (in metric tunes) of an aircraft. The last code-call accesses the agent *supplier* and returns the cargo planes that are available at a location.

```
statistics : distance(loc1, loc2)
statistics : authorRange(aircraft)
statistics : authorCapacity(aircraft)
supplier : cargoPlane(loc)
```

Figure 7. Code-calls in the military logistics domain.

Figure 3 illustrates a simple agentized method which mounts a cargo in an airplane provided that the airplane has the adequate range and capacity. Although simple, the capability to access remote information, reason on the different numbers provides prompt and accurate information and may decide between the success and failure of an operation.

D. Planning

In what follows, a state denotes the union of the states of all IMPACT agents.

Definition 15 (Application of agentized operator).
Let an agentized operator $op = \textbf{(AgentOp } h\, \chi_{add}\, \chi_{del}\textbf{)}$ be given. Applying op to a state $\mathcal{O}$, denoted by result$(\mathcal{O}, op)$, results in a new state $\mathcal{O}'$ in which:

- for each code call $\mathcal{S} : f(\mathtt{d_1}, \ldots, \mathtt{d_n})$ in χ_{add}, $f(d_1, \ldots, d_n)$ is added to the state of $\mathcal{S}$;
- for each code call $\mathcal{S} : f(\mathtt{d_1}, \ldots, \mathtt{d_n})$ in χ_{del}, $f(d_1, \ldots, d_n)$ is removed from the state of $\mathcal{S}$.

Notice that during the planning process of the A-SHOP algorithm, operators are not applied in the sense of the previous definition. Instead, the monitoring agent keeps track of all operators that are supposed to be applied, without actually modifying the state of the IMPACT agents. When A-SHOP queries for a code call $cc = \mathcal{S} : f(\mathtt{d_1}, \ldots, \mathtt{d_n})$ in χ_{add} to evaluate a method's precondition, 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 $\mathcal{S}$). If the planning process ends successfully, the operators are applied as indicated in the previous definition.

Definition 16 (Plans).A *plan* is a list of heads of ground agentized operator instances. If $P = (p_1 p_2 \dots p_n)$ is a plan and $\mathcal{O}$ is a state, then the *result* of applying P to $\mathcal{O}$ is the state result$(\mathcal{O}, P)$ = result(result$(\dots$(result$(\mathcal{O}, p_1), p_2), \dots), p_n)$.

Definition 17 (Simple reductions).Let t be a task, $\mathcal{O}$ be a state, and m = **(AgentMeth** $h\, \chi\, \mathbf{t}$**)** be an agentized method. Suppose that u is a unifier for h and t, and that v is a unifier that unifies each atom in χ^u with some atom in $\mathcal{O}$. Then the agentized method instance $(m^u)^v$ is *applicable* to t in $\mathcal{O}$, and the result of applying it to t is the task list $\mathbf{r} = (\mathbf{t}^u)^v$. The task list $\mathbf{r}$ is a *simple reduction* of $\mathbf{t}$ by m in $\mathcal{O}$.

Definition 18 (Domains and problems).
A *domain representation* is a set of agentized operators and methods. A *planning problem* is a triple $(\mathcal{O}, \mathbf{t}, \mathcal{D})$, where $\mathcal{O}$ is a state, $\mathbf{t}$= $(t_1 t_2 \dots t_k)$ is a task list, and $\mathcal{D}$ is a domain representation. Suppose $(\mathcal{O}, \mathbf{t}, \mathcal{D})$ is a planning problem and $P = (p_1 p_2 \dots p_n)$ is a plan. Then we say that P solves $(\mathcal{O}, \mathbf{t}, \mathcal{D})$, or equivalently, that P *achieves* $\mathbf{t}$ from $\mathcal{O}$ in $\mathcal{D}$ (we will omit the phrase "in $\mathcal{D}$" if the identity of $\mathcal{D}$ is obvious) if any of the following is true:

Case 1: $\mathbf{t}$ and P are both empty, (i.e., $k = 0$ and $n = 0$);

Case 2: t_1 is a primitive task, p_1 is a simple plan for t_1, $(p_2 \dots p_n)$ achieves $(t_2 \dots t_k)$ from result$(\mathcal{O}, p_1)$;

Case 3: t_1 is a compound task, and there is a simple reduction $(r_1 \dots r_j)$ of t_1 in $\mathcal{O}$ such that P achieves $(r_1 \dots r_j t_2 \dots t_k)$ from $\mathcal{O}$.

The planning problem $(\mathcal{O}, \mathbf{t}, \mathcal{D})$ is *solvable* if there is a plan that solves it.

E. Proof of the Main Theorem

Theorem 2 (Soundness of A-SHOP).
Let $\mathcal{O}$ be a state, $\mathcal{D}$ be a collection of agentized methods and operators and let $\mathbf{t}$ be a list of tasks. Let all the preconditions in the agentized methods and add and delete-lists in the agentized operators are strongly safe wrt. the respective

variables. Furthermore suppose one of the nondeterminstic traces of A-SHOP $(\mathcal{O}, \mathbf{t}, \mathcal{D})$ returns a plan P. Then P solves the planning problem $(\mathcal{O}, \mathbf{t}, \mathcal{D})$.

Proof: We first notice that steps 7 and 12 always terminate: this follows from Lemma 6 (it is exactly where we need the assumptions about strongly safeness of the operators involved). All other steps terminate trivially.

The proof is by induction on n, where n is the number of times A-SHOP is called.

Base case ($n = 1$): In this case A-SHOP does not call itself recursively, so it must return at step 1.

Thus $\mathbf{t} =$ nil and $P =$ nil, so from Case 1 of the definition of "achieves", P achieves t in $\mathcal{O}$.

Induction step: Let $n > 1$, and suppose that the theorem is true for every $m < n$. There are two cases:

Case 1. A-SHOP returns P at step 5. Let $\mathbf{t}, R$,and q be as computed in steps 2–4 of A-SHOP. Then $R = (t_1 t_2 \dots t_i)$ for some i. Let $(p_1 p_2 \dots p_j)$ be the plan returned by the recursive call to A-SHOP $(R, \mathcal{D})$ in step 5. From the induction assumption it follows that $(p_1 p_2 \dots p_j)$ achieves $(t_1 t_2 \dots t_i)$ in the state result$(\mathcal{O}, p)$. But t is a primitive task and p is a simple plan for t in $\mathcal{O}$. Thus from Case 2 of the definition of "achieves," the plan $(p p_1 p_2 \dots p_j)$ achieves the task list $\mathbf{t} = (t t_1 t_2 \dots t_i)$ in $\mathcal{O}$.

Case 2. A-SHOP returns P at step 8. Let t and R be as computed in step 2 of A-SHOP. Then $R = (t_1 t_2 \dots t_i)$ for some i. Let R' be the simple reduction of $t$chosen in step 7 of A-SHOP. We know that $R' = (r_1 r_2 \dots r_j)$ for some j and $P = (p_1 p_2 \dots p_k)$ for some k. From the induction assumption we know that $(p_1 p_2 \dots p_k)$ achieves $(r_1 r_2 \dots r_j t_1 t_2 \dots t_i)$ in $\mathcal{O}$. Thus from Case 3 of the definition of "achieves," P achieves $\mathbf{t} = (t t_1 t_2 \dots t_i)$ in $\mathcal{O}$. ∎

Theorem 3 (Completeness of A-SHOP).
Suppose that the planning problem $(\mathcal{O}, \mathbf{t}, \mathcal{D})$ is solvable, i.e. there exists a sequence of ground instantiations of operators $h_1, \dots, h_n$ from $\mathcal{D}$ satisfying the task list $\mathbf{t}$. Then at least one of the nondeterministic traces of A-SHOP$(\mathcal{O}, \mathbf{t}, \mathcal{D})$ returns a plan.

Proof: As in the last theorem, we notice that steps 7 and 12 in A-SHOP always terminate (Lemma 6). All other steps terminate trivially.

For every plan P that solves $(\mathcal{O}, \mathbf{t}, \mathcal{D})$, let P's *solution depth* be the length of P plus the total number of simple reductions needed to produce P from $\mathbf{t}$. Let the *minimum solution depth* of $(\mathcal{O}, \mathbf{t}, \mathcal{D})$ be the smallest solution depth of any plan that solves it. The proof is by induction on n, where n is the minimum solution depth of $\mathbf{t}$.

Base case ($n = 0$): In this case, $\mathbf{t} =$ nil and $P =$ nil , so A-SHOP returns P at step 1.

Induction step: Let $n > 1$, and suppose that the theorem is true for every $m < n$. There are two cases:

Case 1. $\mathbf{t} = (t_1 t_2 \dots t_k)$ for some k, and t_1 is primitive. Then there must be at least one simple plan p for t_1 for which the minimum solution depth of $(\text{result}(\mathcal{O}, p), (t_2 \dots t_k), \mathcal{D})$ is $n-1$, for otherwise the minimum solution depth of $(\mathcal{O}, \mathbf{t}, \mathcal{D})$ could not be n. At step 6, one of the nondeterministic traces of A-SHOP recursively invokes A-SHOP$(\text{result}(\mathcal{O}, p), (t_2 \dots t_k), \mathcal{D})$. From the induction assumption, this recursive invocation of A-SHOP returns a plan $(p_1 p_2 \dots p_k)$. Thus at step 6, A-SHOP returns $(p p_1 p_2 \dots p_k)$.

Case 2. $\mathbf{t} = (t_1 t_2 \dots t_k)$ for some k, and t_1 is non-primitive. Then there must be at least one simple reduction $R = (r_1 r_2 \dots r_j)$ for t_1 such that the minimum solution depth of $(\mathcal{O}, (r_1 r_2 \dots r_j t_2 \dots t_k), \mathcal{D})$ is $n - 1$, for otherwise the minimum solution depth for $(\mathcal{O}, \mathbf{t}, \mathcal{D})$ could not be n. At step 11, one of the nondeterministic traces of A-SHOP recursively invokes A-SHOP$(\mathcal{O}, (r_1 r_2 \dots r_j t_2 \dots t_k), \mathcal{D})$. From the induction assumption, this recursive invocation of shop returns a plan $(p_1 p_2 \dots p_k)$. Thus at step 6, A-SHOP returns $(p_1 p_2 \dots p_k)$. ∎