IMPACTing SHOP: Planning in a Multi-Agent Environment

Jürgen Dix[‡] and Héctor Muñoz-Avila[†], and Dana S. Nau[†]
‡University of Koblenz
Rheinau 1, 56075 Koblenz, Germany
dix@uni-koblenz.de

[†]University of Maryland
College Park, MD 20742, USA
{munoz,nau}@cs.umd.edu

Abstract

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.

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.

[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.

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 describe a formalism for addressing all of the above listed challenges simultaneously, by integrating the SHOP planning system with the IMPACT multi-agent environment. More precisely, 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:

- 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 6 and conclude with Section 7. The appendix contains detailed definitions of those concepts that could not be covered in the main part of the paper.

2 HTN Planning

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.

HTN planning systems have knowledge bases containing *methods*. Each method is a prescription for how to decompose some task into a set of subtasks, with various restrictions that must be satisfied in order for the method to be applicable, and various constraints on the subtasks and the relationships among them. 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 a task to accomplish, the planner 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 or the interactions among them prevent the plan from being feasible, the planning system will backtrack and try other methods.

As an example, Figure 1 shows two methods for the task of traveling from one location to another: traveling by air, and traveling by taxi. Traveling by air involves the subtasks of purchasing a plane ticket, traveling to the local airport, flying to an airport close to our destination, and traveling from there to our destination. Traveling by taxi involves the subtasks of calling a taxi, riding in it to the final destination, and paying the driver. Each method has restrictions on when it can be used: air travel is only applicable for long distances, and travel by taxi is only applicable for short distances.

Now, consider the task of traveling 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 traveling from the University of Maryland to BWI and traveling from Logan to MIT, we can use the *travel by taxi* method to produce additional subtasks as shown in Figure 1.

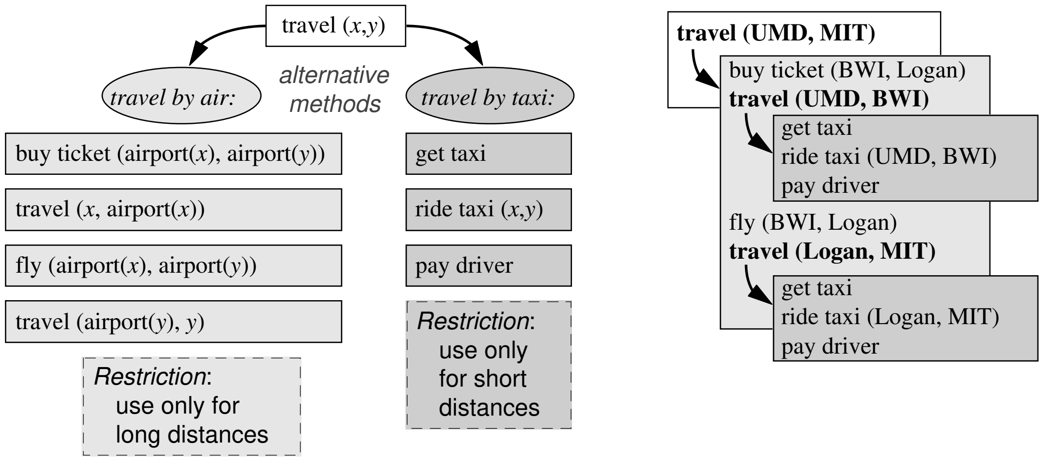

Figure 1: Travel planning example.

Solving a planning problem using HTN planning is generally much more complicated than in this simple example. Here are some of the complications that can arise:

- 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 one will arrive at the airport in time to catch the plane.
- 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, it may be necessary to backtrack and try another method instead.

Our approach is based on a domain-independent HTN planning-algorithm called SHOP (Simple Hierarchical Ordered Planner) [NCLMA99]. This algorithm has the following characteristics:

1. SHOP plans for tasks in the same order that they will later be executed. This avoids some goal-interaction issues that arise in other HTN planners, so that the planning algorithm is relatively simple.
2. Since SHOP knows the complete world-state at each step of the planning process, it can use highly expressive domain representations. For example, it can do planning problems that require complex numeric computations or calls to external programs.

3. SHOP's expressive power makes it possible to encode highly efficient domain-dependent planning algorithms into SHOP. For example, in our experimental tests on a standard suite of planning problems [NCLMA99], SHOP solved the problems orders of magnitude faster than the Blackbox planner [KS99], and several times as fast as the TLplan planner [BK00].

SHOP's expressive power also makes it ideal as a basis for integrating planning with external information sources, such as the IMPACT framework as described in this paper.

3 The Multi-Agent System 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:

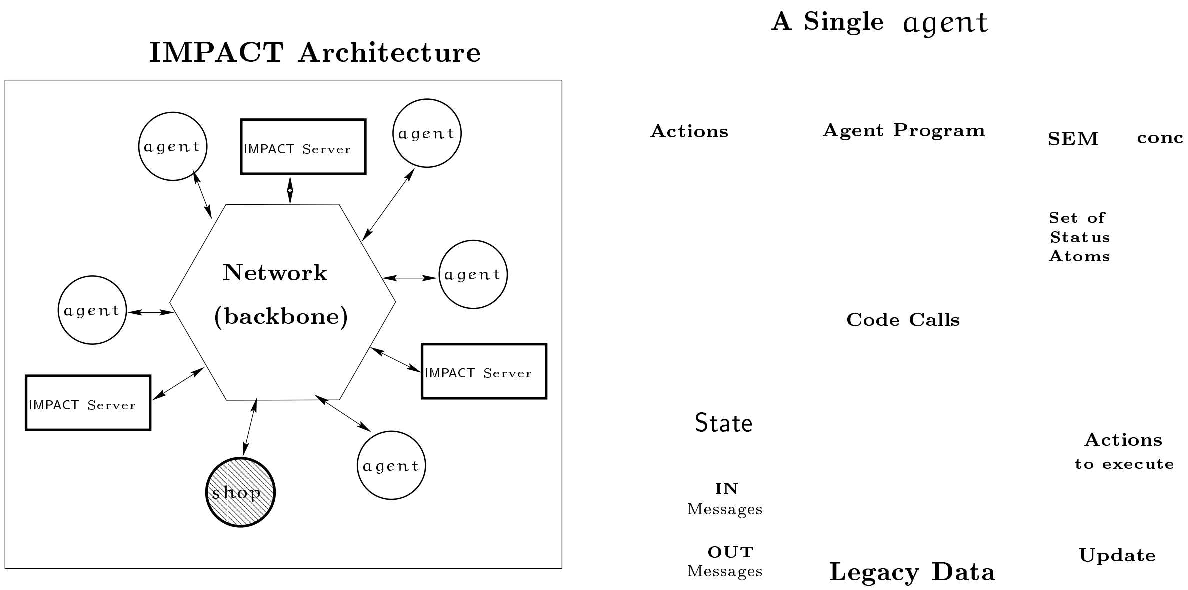

Figure 2: (1) SHOP as a planning agent in IMPACT. (2) An Agent in IMPACT.

- A 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: (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 actions its *status* (permitted, obliged, forbidden, ...). The agents 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.

Suppose we consider a body $\mathcal{S} = (\mathcal{T}_{\mathcal{S}}, \mathcal{F}_{\mathcal{S}})$ of software code that an arbitrary agent is built on top of. $\mathcal{F}_{\mathcal{S}}$ 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 objects the agent currently has—the types of these objects must be in the base set of types in $\mathcal{T}_{\mathcal{S}}$ (note that the message box is just one of those types). 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 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.*

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 contained in a certain set or not. To do this, weintroduce 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.

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 just in case 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 nothing more than a conjunction of atomic code calls, with some 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* $\mathtt{s},\mathtt{t}$ *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.*

We are know 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.

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).

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 consists 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 illustrattes 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.

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 enquiry 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 task list.*

The definition of agentized method differs from the definition of method in SHOP in that the preconditions are IMPACT's code call conditions.

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 code call conditions (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.*

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 2 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 capability* (in metric tones) of an aircraft. 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.

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 of tasks to achieve, A-SHOP looks at the first task in the list. If that task is primitive (Step

procedure *A-SHOP*($\mathbf{t}, \mathcal{D}$)
1. **if** $\mathbf{t} = nil$ **then return** *nil*
2. $\mathbf{t}$:= the first task in $\mathbf{t}$; R := the remaining tasks
3. **if** $\mathbf{t}$ is primitive and a simple plan for t exists **then**
4. $q := simplePlan(t)$
5. **return** $cons(q,$ A-SHOP$(R, \mathcal{D}))$
6. **else if** $\mathbf{t}$ is non-primitive and there is a reduction of t **then**
7. **nondeterministically** choose a reduction. That is, nondeterministically choose an agentized method, (: **AgentMeth** $h\,\chi\,\mathbf{t}$), with μ the most general unifier of h and $\mathbf{t}$ and substitution θ in IMPACT : $instances(\chi)$ such that $\mathbf{t}\mu\theta$ is reducible
8. **return** *A-SHOP*($append(\mathbf{t}\mu\theta, R), \mathcal{D}$)
9. **else return** *FAIL*
10. **end if**

end *A-SHOP*

procedure $simplePlan(\mathbf{t})$

11. **nondeterministically** choose an agentized operator $op =$ (: **AgentOp** $h\,\chi_{add}\,\chi_{del}$) with ν the most general unifier of h and $\mathbf{t}$
12. IMPACT : $apply(op\,\nu)$
13. **return** $op\,\nu$

end *A-SHOP*

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

3), then A-SHOP looks for a *simple plan* consisting of a single operator (Step 4), and then calls itself recursively to examine the remaining tasks (Step 5). If the task is compound and reducible (step 6), then A-SHOP applies an agentized method to produce subtasks (steps 7-8).

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. The main differences between SHOP and A-SHOP are as follows

- A-SHOP uses IMPACT's code call mechanism to evaluate the preconditions of each agentized method and return all possible instantiations of the method (Step 7);
- A-SHOP uses agentized operators to modify the states of the IMPACT agents (Step 12).

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).

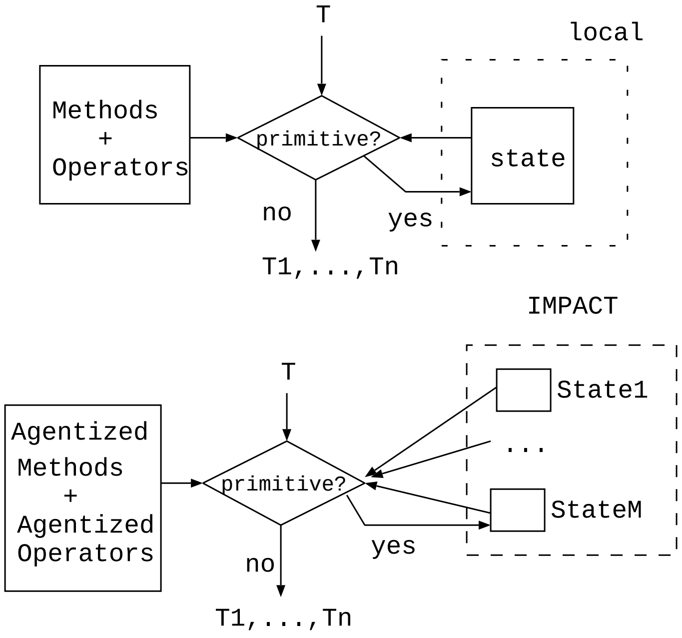

Figure 5: Data flow in (a) SHOP and (b) A-SHOP.

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 $\mathtt{math}$ which provides several functions to deal with integers. The code call $\mathtt{math}:geq(\mathtt{X})$ enumerates all integers greater or equal to $\mathtt{X}$. This cc is not executable, because it is not ground. Only if the variable $\mathtt{X}$ 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

$$\mathbf{in}(\mathtt{X}, \mathtt{math}:geq(\mathtt{25}))\,\&\,\mathbf{in}(\mathtt{Y}, \mathtt{math}:square(\mathtt{X}))\,\&\,Y \leq 2000,$$

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; 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.

Theorem 1 (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, 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, the problem of applying the agentized operator to $\mathcal{O}$ can be algorithmically solved.*

Theorem 2 (Soundness and 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, then A-SHOP is correct and complete.*

6 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 [Koe98, 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+ng, 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.

7 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`.

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

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

  which is essentially a symbolic process. On the other hand, some of its preconditions are numerical comparisons (i.e., $\mathtt{Dist} \leq \mathtt{DCargoPL}$). 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

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

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

[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.

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

[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.

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

[Koe98] J. Koehler. Planning under Resource Constraints. In *Proceedings of the 13th European Conference on Artificial Intelligence, pp 489-493*, 1998.

[KS99] H. Kautz and B. Selman. Unifying sat-based and graph-based planning. In *IJCAI-99*, 1999.

[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.

[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+ng] 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*, forthcoming.

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

[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.

[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.

[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.

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 6 (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 7 (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 $\mathtt{X}_1, \ldots, \mathtt{X}_n$*);*

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 8 (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 9 (Agent Program) *An agent program* $\mathcal{P}$ *is a finite set of rules of the form*

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

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 Soundness and Completeness

Note that the following definitions underlying strongly safeness are important for the general case, when an agent is build 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 10 (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 $\mathcal{S}:f(-,\mathtt{5},-)$ 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 11 (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 12 (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 13 (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 6 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 tones) 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 6: 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 Sketch of Proof

Theorem 3 *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, then:*

- A-SHOP is correct. *Suppose one of the nondeterministic traces of ASHOP(S,T,D) returns a plan P. Then P solves the planning problem (S,T,D).*
- A-SHOP is complete. *Suppose that the planning problem (S,T,D) is solvable. Then at least one of the nondeterministic traces of ASHOP(S,T,D) returns a plan.*
- A-SHOP is complete.

Idea of the proof. The proof is done in two parts: first we proof that both soundness and completeness of A-SHOP holds assuming that the calls to IMPACT to evaluate the preconditions and to modify the state terminate (Steps 7 and 10 in Figure 4 respectively). Second, we conclude that since the preconditions in the agentized methods and add and delete-lists in the agentized operators are strongly safe, termination of Steps 7 and 10 in Figure 4 is a given.

The proof of soundness of A-SHOP modulo termination of Steps 7 and 10 follows by induction on the number of calls to the A-SHOP procedure. Given a task list **t** the proof distinguishes whether the first element, t, in **t** is primitive or not. If it is primitive it shows the simple plan which effectively solves t and if it is compound, it shows that the solution for the reduction of t is the same as for **t**. The argument concludes by observing the induction hypothesis. The proof of completeness is similar but this time the induction is on the minimum solution deph for the set of tasks **t**. We omit further details at this point as the details are cumbersome.