IMPACTing SHOP: Foundations for integrating HTN Planning and Multi-Agency

Héctor Muñoz-Avila†, Jürgen Dix‡, Dana S. Nau† and Yue Cao†
†Department of Computer Science
University of Maryland
College Park, MD 20742, USA
‡Department of Computer Science
University of Koblenz
Rheinau 1, D-56075 Koblenz, Germany

February 3, 2000

Abstract

Keywords: HTN-planning, multi-agents

In this paper we describe a formalism for integrating the *SHOP* HTN planning system with the *IMPACT* multi-agent environment. Our formalism provides 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 this agentized version of *SHOP* will preserve soundness and completeness 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:

- *The need to interact with external information sources* [Chien et al., 1995]. The problem tends to be complicated by the fact that frequently the information sources are 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 [Benton and Subrahmanian, 1994], a variety of US Army logistics data including specialized Oracle and nested multi-record TAADS data [Schafer et al., 1998], a variety of US Army simulation data from a massive program called JANUS, Training and Instrumentation Command, a face recognition program, and so forth.

- *The need to perform mixed symbolic/numeric reasoning.* For example, [Nau et al., 1998] 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.

- *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: for example, 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.

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.

In this paper we describe a formalism that addresses all of the above challenges simultaneously, by integrating the *SHOP* planning system with the *IMPACT* multi-agent environment. *SHOP* [Nau et al., 1999] is a very efficient HTN planner. *IMPACT* [Subrahmanian et al., 2000, Eiter et al., 1999] 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. 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.

Our work includes the following:

- a definition of the *A-SHOP* planning algorithm, an agentized version of *SHOP* that runs in the *IMPACT* environment;

- 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;

- an implementation of this formalism (work currently in progress).

2 Related Work

Several AI planning systems (most notably HTN planning systems such as SIPE [Wilkins, 1988] and O-Plan [Currie and Tate, 1991], and *SHOP* [Nau et al., 1999]) have the ability to evaluate numeric conditions as attached procedures. However, the lack of a formal semantics for these attached procedures makes it 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 [Koehler, 1998, Kautz and Walser, 1999, Wolfman and Weld, 1999]. 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 [Etzioni et al., 1992, Golden et al., 1994, Knoblock, 1996, Friedman and Weld, 1997]. For example, a modified version of UCPOP uses *information gathering goals* to extract information from the external information sources [Knoblock, 1996]. 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 has been the focus of research for many years (e.g., [Davis and Smith, 1983]). With the advances in agent research (see for example [Wooldridge and Jennings, 1995]), 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 [desJardins et al., 1999] for a recent survey).

3 HTN Planning with Ordered Task Decompositions: *SHOP*

Below we formalize the idea of ordered task decomposition that forms the basis of *SHOP*'s planning Hierarchical Task Network (HTN) algorithm. HTN planning is a planning paradigm frequently used in applied research projects [Wilkins, 1988, Currie and Tate, 1991]. One of the main reasons for the use of HTN planning is its expressiveness, which allows it to encode not only knowledge about the domain but also problem-solving strategies. HTN planning has been shown to be more expressive than STRIPS planning [Erol et al., 1994]. Recently, an HTN planning system, *SHOP*, has been developed that uses a new approach to HTN planning called *ordered-task decomposition*. *SHOP* outperforms several planning systems by orders of magnitude [Nau et al., 1999].

Definition 1 (State) *A* state $\mathcal{O}$ *is a finite set of ground logical atoms. A* conjunct *is a finite set of logical atoms. A state* $\mathcal{O}$ *satisfies a conjunct* C modulo a substitution ν *if, by definition, all ground instances of* $C\nu$ *are subsets of* $\mathcal{O}$.

Definition 2 (Primitive and Compound Tasks) *A task is a list of the form* $(s\ t_1, ..., t_n)$*, where* s *is a symbol, called the* task name*, and* t_1, ..., t_n *are terms,* *called the* task's arguments. *A task is* primitive *if its task name is a primitive task symbol Conversely, a task is* compound *if its task name is a compound task symbol . abbreviated by* $\mathbf{t}$*, is a list of tasks.*

Definition 3 (Operator) *An* operator *is an expression (*: **Op** *h Add Del), where h (the* head*) is a primitive task and Add and Del are conjuncts (called the* add *and* delete *lists). The set of variables in the tasks in Del and Add is a subset of the set of variables in h.*

An *operator* indicates how the state is modified when an operator *is applied*: if ν is a substitution such that $h\nu$ is ground, the state, $h^{\nu}(\mathcal{O})$, resulting from applying the operator to the state $\mathcal{O}$, is defined by $h^{\nu}(\mathcal{O}) =_{def} (\mathcal{O} - Del\,\nu) \cup Add\,\nu$ (i.e., removing every atom in *Del* and adding every atom in *Add*).

Definition 4 (Method) *A* method *is an expression (*: **Meth** *h Pre* $\mathbf{t}$*), where h (the method's* head*) is a compound task, Pre (the method's* preconditions*) is a conjunct and* $\mathbf{t}$ *is a task list.*

A method expression has the following meaning: to solve a compound task $\mathbf{t}$ matching h with a substitution μ it suffices to solve the tasks in $t\mu$ provided that the method is *applicable* to the current state, $\mathcal{O}$. Formally, a method, m, is *applicable* to $\mathbf{t}$ relative to $\mathcal{O}$ if h matches $\mathbf{t}$ with a substitution μ and $\mathcal{O}$ satisfies Preμ. We denote with App$(\mathbf{t}, \mathcal{O})$, the set of all applicable methods to $\mathbf{t}$ relative to $\mathcal{O}$.

Definition 5 (ordered reduction tasks lists) *The set of* ordered reduction tasks lists*,* $\mathit{red}(\mathbf{t}, \mathcal{O})$*, is defined as follows:*

$$\mathit{red}(\mathbf{t}, \mathcal{O}) = \bigcup_{m \in App(\mathbf{t}, \mathcal{O})} \mathit{red}_{simple}(\mathbf{t}, \mathcal{O}, m)$$

where $\mathit{red}_{simple}(\mathbf{t}, \mathcal{O}, m)$ *is a* simple ordered reduction *of* $\mathbf{t}$ *by* m *relative to* $\mathcal{O}$*. If* $\mathbf{t} = (t_1 ... t_n)$ *is a task list and* t_j *is the first compound task in* $\mathbf{t}$*, we define* $\mathit{red}_{simple}(\mathbf{t}, \mathcal{O}, m)$ *as the set of all tasks lists of the form* $(t_1 ... t_{j-1} r_j t_{j+1} ... t_n)\mu\nu$ *where* r_i *are the subtasks of* m*. The substitution* μ *is the most general unifier of the head of* m *and* $\mathbf{t}$ *and the substitution* ν *is such that the precondition of* m *satisfies the current state with* ν*. The operation of replacing* $\mathbf{t}$ *with an element of* $\mathit{red}_{simple}(\mathbf{t}, \mathcal{O}, m)$ *is called an* application *of the method* m *and replacing* $\mathbf{t}$ *with an element of* $\mathit{red}(\mathbf{t}, \mathcal{O})$ *is called an* ordered reduction *or* ordered decomposition *of* $\mathbf{t}$*.*

We use the term reduction (decomposition) instead of ordered reduction (ordered decomposition). The definition of ordered reduction tasks lists formalizes the notion of ordered task decomposition; essentially, the first task in the task list will be decomposed to primitive tasks, then the second one and so forth. Ordered task decompositions are an especial form of HTNs as defined in [Erol et al., 1994]; in ordered task decompositions the tasks in the task lists are totally ordered and no truth constraints are defined.[1]

[1] A truth constraint indicates condition commitments such as (t, l, t') indicating that the condition l needs to be true between the tasks t and t'.

Definition 6 (Plan, Planning Problem) *A planning problem is a triple $(\mathcal{O}, \mathbf{t}, \mathcal{D})$, where $\mathcal{O}$ is a state, $\mathbf{t}$ is a task list, and $\mathcal{D}$ is the list of methods and operators, called the domain. A* plan *σ is a completion of $\mathbf{t}$ at $\mathcal{O}$, denoted by $comp(\mathbf{t}, \mathcal{O}, \mathcal{D})$, if σ is a total ordering of the ground instances of $\mathbf{t}$, provided that $\mathbf{t}$ is a primitive task list (i.e., if all tasks in $\mathbf{t}$ are primitive). Otherwise, $comp(\mathbf{t}, \mathcal{O}, \mathcal{D})$ is the empty set $\emptyset$.*

We can now define the set of all valid solution plans, $\mathsf{sol}(\mathbf{t}, \mathcal{O}, \mathcal{D})$, as characterized in [Erol et al., 1994]:

Definition 7 (Solutions $\mathsf{sol}(\mathbf{t}, \mathcal{O}, \mathcal{D})$) *Let $\mathcal{D}$ be a planning domain, $\mathcal{O}$ an initial state and $\mathbf{t}$ a task list. $\mathsf{sol}_0(\mathbf{t}, \mathcal{O}, \mathcal{D})$ is defined inductively using the following two rules*

Rule 1: *If $\sigma \in comp(\mathbf{t}, \mathcal{O}, \mathcal{D})$ then $\sigma \in \mathsf{sol}(\mathbf{t}, \mathcal{O}, \mathcal{D})$.*

Rule 2: *If $\mathbf{t}' \in red(\mathbf{t}, \mathcal{O}, \mathcal{D})$ and $\sigma \in comp(\mathbf{t}', \mathcal{O}, \mathcal{D})$ then $\sigma \in \mathsf{sol}(\mathbf{t}, \mathcal{O}, \mathcal{D})$.*

The set of plans $\mathsf{sol}(\mathbf{t}, \mathcal{O}, \mathcal{D})$ is defined as follows:

$$\mathsf{sol}_0(\mathbf{t}, \mathcal{O}, \mathcal{D}) = \{()\},$$

$$\mathsf{sol}_1(\mathbf{t}, \mathcal{O}, \mathcal{D}) = comp(\mathbf{t}, \mathcal{O}, \mathcal{D})$$

$$\mathsf{sol}_{n+1}(\mathbf{t}, \mathcal{O}, \mathcal{D}) = \bigcup_{\mathbf{t}' \in red(\mathbf{t}, \mathcal{O}, \mathcal{D})} \mathsf{sol}_n(\mathbf{t}', \mathcal{O}, \mathcal{D})$$

$$\mathsf{sol}(\mathbf{t}, \mathcal{O}, \mathcal{D}) = \bigcup_{n<\omega} \mathsf{sol}_n(\mathbf{t}, \mathcal{O}, \mathcal{D})$$

Now we present the *SHOP* ordered task decomposition algorithm. We then show that it is sound and complete wrt. Definition 7.

The *SHOP* algorithm is depicted in Figure 1. The first subtask is analyzed (step 2). If the task is primitive (step 3), a simple plan is obtained if possible (steps 4 and 5) and the process continues with the remaining tasks (step 6). If the task is compound (step 8), sub-tasking is performed if possible (steps 9-14). The procedure *setSimpleReductions* computes the set of ordered simple reductions of $\mathbf{t}$ in $\mathcal{O}$. The procedure *simplePlan* finds an applicable operator and applies it.

To prove soundness and completeness of the *SHOP* algorithm, we will show that $\mathsf{sol}(\mathbf{t}, I, \mathcal{D})$ is equal to the set of all plans generated by *SHOP*, $\mathsf{sol}_{SHOP}(\mathbf{t}, I, \mathcal{D})$.

Theorem 1 (Soundness) *SHOP is sound, i.e.*

$$\mathsf{sol}_{SHOP}(\mathbf{t}, \mathcal{O}, \mathcal{D}) \subseteq \mathsf{sol}(\mathbf{t}, \mathcal{O}, \mathcal{D}).$$

Proof: We will show by induction that if $\theta \in \mathsf{sol}_{SHOP}(\mathbf{t}, \mathcal{O}, \mathcal{D})$ holds and θ is generated in n iterations, then $\theta \in \mathsf{sol}_n(\mathbf{t}, \mathcal{O}, \mathcal{D})$.

```
procedure find-Plan(O, t, D)
  return seek-plan(O, t, D, nil)
end find-Plan

procedure seek-Plan(O, t, D, p)
  1. if t = nil then return the list (p)
  2. t:= the first task in t; R := the remaining tasks
  3. if t is primitive then
    4. q := simplePlan(t, O)
    5. if q ≠ FAIL
      6. return seek-plan(q(O), R, D, append(p, q))
    7. else return FAIL
  8. else
    9. Reds = setSimpleReductions(t, O)
    10. for every r in Reds
      11. ans = seek-plan(O, append(r, R), D, p)
      12. if ans ≠ FAIL then return ans
    13. end for
  14. return FAIL
end if
end
```

Figure 1: SHOP's ordered task decomposition algorithm.

Base step (n = 0): Thus, $\mathbf{t} = \emptyset$ and *SHOP* returns the empty plan (step 1), which is in $\mathsf{sol}_0(\mathbf{t}, \mathcal{O}, \mathcal{D})$.

Induction step (n = k+1): Let θ the plan returned by *SHOP* after $n+1$ iterations for $\mathbf{t}$. Let $\mathbf{t}$ and R be the values of the assignments in step 2 at iteration 1 of *SHOP*.

> **Case 1.** if $\mathbf{t}$ is primitive, then $\theta = q\theta'$, where q is a simple plan for $\mathbf{t}$ (steps 3-6) and θ' is a plan for R. By induction hypothesis $\theta' \in \mathsf{sol}_k(R, \mathcal{O}, \mathcal{D})\mathsf{sol}_k(\mathbf{R}, \mathcal{O}, \mathcal{D})$ and because $\mathbf{t}$ is primitive $\{q\} = comp(t, \mathcal{O}, \mathcal{D}) = \mathsf{sol}_1(\mathbf{t}, \mathcal{O}, \mathcal{D})$. Thus, $q\theta' \in \mathsf{sol}_{k+1}(\mathbf{R}, \mathcal{O}, \mathcal{D})$.
>
> **Case 2.** If $\mathbf{t}$ is compound, then let *ans* be the successful plan meeting the condition of step 12 and let r be the corresponding reduction. Because *ans* is generated in k steps and *ans* is in $\mathsf{sol}_{SHOP}(\mathbf{R}, \mathcal{O}, \mathcal{D})$, then, $ans \in \mathsf{sol}_k(\mathbf{R}, \mathcal{O}, \mathcal{D})$ holds. Thus, $ans \in \mathsf{sol}_{k+1}(R, \mathcal{O}, \mathcal{D})$. Clearly, *ans* is equal to θ because $\mathsf{sol}_{SHOP}((\mathbf{tR}), \mathcal{O}, \mathcal{D})$ is equal to $\mathsf{sol}_{SHOP}((\mathbf{rR}), \mathcal{O}, \mathcal{D})$.

Theorem 2 (Completeness) *SHOP is complete, i.e.*

$$sol(\mathbf{t}, \mathcal{O}, \mathcal{D}) \subseteq sol_{SHOP}(\mathbf{t}, \mathcal{O}, \mathcal{D}).$$

```
procedure setSimpleReductions(t, O)
  Assign the empty set to Res
  for every method m, (: Meth h Pre t), with μ the
      most general unifier of h and t and m applicable
      to t relative to O
    for every substitution θ in instances(Pre, O)
      add tμθ into Res
    end for
  end for
  return Res
end setSimpleReductions
```

Figure 2: Algorithm for simple reductions.

```
procedure simplePlan(t, O)
  if there is an operator op = (: Op h D A) with ν
     the most general unifier of h and t then
     apply(op ν, O)
     return op ν
  else
    return FAIL
  end if
end simplePlan
```

Figure 3: Algorithm for simple plan.

Proof: We will show by induction that if $\theta \in \mathsf{sol}_n(\mathbf{t}, \mathcal{O}, \mathcal{D})$ holds, then θ is generated by *SHOP* in at most n iterations.

Base step (n = 0): Thus, θ is the empty plan which is generated in 0 iterations of *SHOP* (step 1).

Induction step (n = k+1): Let θ be a plan in $\mathsf{sol}_{k+1}(\mathbf{t}, \mathcal{O}, \mathcal{D})$.

Case 1. If θ is in $\mathsf{sol}_k(\mathbf{t}, \mathcal{O}, \mathcal{D})$, θ is generated by *SHOP* in at most k steps by induction hypothesis.

Case 2. If θ is not in $\mathsf{sol}_k(\mathbf{t}, \mathcal{O}, \mathcal{D})$, there exists $\mathbf{t}'$ in $\mathsf{red}(\mathbf{t}, \mathcal{O}, \mathcal{D})$ such that θ is in $\mathsf{sol}_k(\mathbf{t}', \mathcal{O}, \mathcal{D})$. By induction hypothesis θ is generated in at most k steps by *SHOP* to solve d'. Let $\mathbf{t} = (tR)$, where $\mathbf{t}$ is a compound task reduced with $\mathbf{t}'$. Thus, $\mathbf{t}'$ must be one of the simple reductions for $\mathbf{t}$ generated at step 9. As a result, θ is one of the possible answers of step 11 meeting the condition of step 12 and, thus, an answer of *SHOP* at $k + 1$ iterations for $\mathbf{t}$.

Notice that in the soundness and completeness of the *SHOP* algorithm we implicitly assumed that the centralized state requirement was satisfied: this assumption was present in the use of the function *instances*$(Pre, \mathcal{O})$ in Figure 2 as well as for applying an operator to transform the state (i.e., $apply(op\,\nu, \mathcal{O})$ in Figure 3). This requirement is clearly satisfied in typical AI planning systems where the instances are found by examining the state, which is stored in local memory together with the planning algorithm. In our framework, however, the knowledge about the state is distributed between several information sources that are not even required to use a uniform language. Thus, we need to re-define how *SHOP* evaluates its conditions and applies its operators. More concretely, in the section after the next, we will state conditions to guarantee that $instances(Pre, \mathcal{O})$ and $apply(op\,\nu, \mathcal{O})$ can be evaluated.

4 An Architecture for Agent Integration: *IMPACT*

Most existing work on agents has assumed a purely logical description of agents. This is very similar to most planning systems: the underlying representation language is often first order logic (or a subset thereof) and other data can not be incorporated easily.

The *IMPACT* project (see `http://www.cs.umd.edu/projects/impact/`) [Subrahmanian et al., 2000, Eiter et al., 1999, Eiter and Subrahmanian, 1999]) aims at developing a powerful and flexible, yet easy to handle framework for the interoperability of distributed heterogeneous sources of information. A methodology for transforming arbitrary software (legacy code) into an *agent* has been developed. Agents act in their environment according to a well defined decision policy. The actions an agent can take and the program to encode the agent's behavior are specified by the user through the *IMPACT Agent Development Environment*. The main cycle an *IMPACT* agent goes through depends on the notion of *state* of an agent. An *IMPACT* agent takes actions based on its current state, its agent program and a chosen semantics. It then executes these actions concurrently according to a specified notion of concurrency.

IMPACT agents are built on top of arbitrary software code. We view such code formally as a triple $\mathcal{S} =_{def} (\mathcal{T}_{\mathcal{S}}, \mathcal{F}_{\mathcal{S}}, \mathcal{C}_{\mathcal{S}})$ where:

1. $\mathcal{T}_{\mathcal{S}}$ is the set of all data types managed by $\mathcal{S}$,
2. $\mathcal{F}_{\mathcal{S}}$ is a set of predefined functions which makes access to the data objects managed by the agent available to external processes, and
3. $\mathcal{C}_{\mathcal{S}}$ is a set of type composition operations. A type composition operator is a partial n-ary function c which takes as input types $\tau_1, \ldots, \tau_n$ and yields as a result a type $c(\tau_1, \ldots, \tau_n)$. As c is a partial function, c may only be defined for certain arguments $\tau_1, \ldots, \tau_n$, i.e., c is not necessarily applicable on arbitrary types.

Intuitively, $\mathcal{T}_{\mathcal{S}}$ is the set of all data types that are managed by the agent. $\mathcal{F}_{\mathcal{S}}$ intuitively represents the set of all function calls supported by the package $\mathcal{S}$'s application programmer interface (*API*). $\mathcal{C}_{\mathcal{S}}$ is the set of ways of creating new data types from existing data types.

Definition 8 (State of an Agent) *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 either in the base set of types in* $\mathcal{T}_{\mathcal{S}}$ *or must be a type obtained by composition using the composition operations in* $\mathcal{T}_{\mathcal{S}}$.

Suppose we consider a body $\mathcal{S} = (\mathcal{T}_{\mathcal{S}}, \mathcal{F}_{\mathcal{S}})$ of software code that an agent is built on top of. For the agent to perform logical reasoning on top of such third party data structures and code, the agent must have a language within which it can reason about the agent state. In this section, we introduce the important concept of a *code call atom*—this is the basic syntactic object used to access multiple heterogeneous data sources.

Given any type $\tau \in \mathcal{T}_{\mathcal{S}}$, we assume that there is a set $Var(\tau)$ of variable symbols ranging over τ. If $\mathtt{X} \in Var(\tau)$ is such a variable symbol, and if τ is a complex record type having fields $\mathtt{f_1}, \ldots, \mathtt{f_n}$, then we require that $\mathtt{X.f_i}$ be a variable of type τ_i where τ_i is the type of field $\mathtt{f_i}$. In the same vein, if $\mathtt{f_i}$ itself has a sub-field g of type γ, then $\mathtt{X.f_i.g}$ is a variable of type γ, and so on. In this case, we call X a *root-variable*, and the variables $\mathtt{X.f_i}$, $\mathtt{X.f_i.g}$, etc. *path-variables*. For any path variable Y of the form X.path, where X is a root variable, we refer to X as the root of Y, denoted by *root*(Y); for technical convenience, *root*(X), where X is a root variable, refers to itself.

Definition 9 (Code Call $\mathcal{S}:f(\mathtt{d_1}, \ldots, \mathtt{d_n})$) *Suppose* $\mathcal{S} =_{def} (\mathcal{T}_{\mathcal{S}}, \mathcal{F}_{\mathcal{S}}, \mathcal{C}_{\mathcal{S}})$ *is some software code and* $f \in \mathcal{F}_{\mathcal{S}}$ *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.*

Intuitively, the syntactic string $\mathcal{S}:f(\mathtt{d_1}, \ldots, \mathtt{d_n})$ may be read as: *execute function* f *as defined in package* $\mathcal{S}$ *on the arguments* $\mathtt{d_1}, \ldots, \mathtt{d_n}$.

We assume that the output signature of any code call is an object of type set.

In general, as we will see later, code calls are executable when they are ground. Thus, non-ground code calls must be *instantiated* prior to attempts to execute them.

A code call according to Definition 9 executes an *API* function and returns as output a set of objects of the appropriate output type. Code-call atoms are *logical atoms* that are layered on top of code-calls. They are defined through the following inductive definition.

Definition 10 (Code Call Atom 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 just in case X is **not** in the set of objects returned by the code call *cc*.

Code call atoms, when evaluated, return boolean values, and thus may be thought of as special types of logical atoms. Intuitively, a code call atom of the form $\mathbf{in}(\mathtt{X}, \mathtt{cc})$ succeeds 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.

We will now 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 11 (Code Call Condition χ) *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.*

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 12 (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* $\mathsf{Sol}(\chi)_{\mathcal{T}_\mathcal{S}, \mathcal{O}_\mathcal{S}}$

Definition 13 (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 this section 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 $i = 1, \ldots, n$.*

5 IMPACTing *SHOP*

SHOP is an efficient HTN planner. *SHOP* works by applying *methods* to *compound tasks* to decompose them into simpler tasks, and keeps decomposing tasks until *primitive tasks* are reached that can be executed directly using *operators* (see [Nau et al., 1999] for details). The integration of *SHOP* with the *IMPACT* multi-agent environment requires three steps:

1. Replace the atoms in *SHOP*'s preconditions, add-lists, and delete-lists with code call conditions.

2. Agentize *SHOP* so that it can communicate with other *IMPACT* agents.

3. Make adjustments to *SHOP*'s planning algorithm.

5.1 Modifying *SHOP*'s Atoms

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. An *agentized method* is an expression $(\mathbf{:\,AgentMeth}\ h\,\chi\,\mathbf{t})$ 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. An *agentized operator* is an expression $(:\textbf{AgentOp}\ h\,\chi_{add}\,\chi_{del})$, 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.

Agentized operators differ from *SHOP*'s operators in that the add- and delete-lists are code call conditions.

Figure 4 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:
$\mathit{AirTransport}(\texttt{LocFrom}, \texttt{LocTo}, \texttt{Cargo}, \texttt{CargoWeight})$

Preconditions:
$\textbf{in}(\texttt{CargoPL}, \mathtt{supplier}:\mathit{cargoPlane}(\texttt{locFrom}))\&$
$\textbf{in}(\texttt{Dist}, \mathtt{statistics}:\mathit{distance}(\texttt{locFrom}, \texttt{locTo}))\&$
$\textbf{in}(\texttt{DCargoPL}, \mathtt{statistics}:\mathit{authorRange}(\texttt{CargoPL}))\&$
$\texttt{Dist} \leq \texttt{DCargoPL}\&$
$\textbf{in}(\texttt{CCargoPL}, \mathtt{statistics}:\mathit{authorCapacity}(\texttt{CargoPL}))\&$
$\texttt{CargoWeight} \leq \texttt{CCargoPL}\&$

Subtasks:
$\mathit{load}(\texttt{Cargo}, \texttt{locFrom})$
$\mathit{fly}(\texttt{Cargo}, \texttt{locFrom}, \texttt{LocTo})$
$\mathit{unload}(\texttt{Cargo}, \texttt{locTo})$

Figure 4: Agentized method for a military logistics problem.

5.2 Agentizing *SHOP*

To agentize *SHOP*, we can use the general-purpose agentizing algorithm described in [Subrahmanian et al., 2000]. This enables *SHOP* to communicate with other *IMPACT* agents and vice-versa. The algorithm basically 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 [Subrahmanian et al., 2000] for a detailed discussion of the algorithm).

5.3 *A-SHOP*: Modifying the *SHOP* Algorithm

On the top level, the *A-SHOP* algorithm is the same as the *SHOP* algorithm (see Figure 1). The first subtask is analyzed (step 2). If the task is primitive (step 3), a simple plan is obtained if possible (steps 4 and 5) and the process continues with the remaining tasks (step 6). If the task is compound (step 8), subtasking is performed if possible (steps 9–14).

There are two changes that must be made to the *SHOP* algorithm at a lower level when evaluating $simplePlan(\mathbf{t}, \mathcal{O})$ (Figure 3) and $setSimpleReductions(\mathbf{t}, \mathcal{O})$ (Figure 2). That is, when applying an operator and evaluating a method's preconditions:

Change to $simplePlan(\mathbf{t}, \mathcal{O})$: $apply(op\,\nu, \mathcal{O})$ now means to apply the changes indicated in the actions. i.e. execute the code of each action (Definition 13) to make sure the add- and delete-lists are taken into account. Note that in the simplest case, such code could be just the update described by

$$h^{\nu}(\mathcal{O}) =_{def} (\mathcal{O} - Del\,\nu) \cup Add\,\nu,$$

whereas it could also be a sophisticated implementation for manipulating the data objects of the software the agent is built upon.

Change to $setSimpleReductions(\mathbf{t}, \mathcal{O})$: the call to $instances(precs(m), \mathcal{O})$ now evaluates a code call condition in *IMPACT* instead of a collection of predicates directly in the state. As explained in the previous section, evaluating a code call condition reduces to evaluating code call atoms of the form $\mathbf{in}(\mathtt{X}, \mathtt{cc})$, where *cc* is a code call accessing external data sources. These code calls are executed only *as needed* and we have to make sure (using appropriate syntactical conditions) that code calls *(1)* can be executed and *(2)* return a finite answer. We explain in the next section how we accomplish this for agents based on arbitrary software $\mathcal{S}$.

Completeness and Correctness of *A-SHOP*. The changes indicated above ensure that the proofs of soundness and correctness for *SHOP* are valid provided that we can establish conditions under which the evaluating *IMPACT*'s actions and code call conditions can be evaluated. This will be discussed later.

6 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 5 shows some of the code-calls for this application. The first three code-calls access the agent $\mathtt{statistics}$ 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

$\mathsf{statistics}: \mathit{distance}(\mathtt{loc1}, \mathtt{loc2})$
$\mathsf{statistics}: \mathit{authorRange}(\mathtt{aircraft})$
$\mathsf{statistics}: \mathit{authorCapacity}(\mathtt{aircraft})$
$\mathsf{supplier}: \mathit{cargoPlane}(\mathtt{loc})$

Figure 5: Code-calls in the military logistics domain.

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.

Figure 4 illustrates a simple agentized method which mounts a cargo in an airplane provided that the airplane has the adequate range and capacity.

7 Evaluating Code-call Conditions and Applying Actions in *IMPACT*

We have seen that the code call mechanism of *IMPACT* transfers data in arbitrary format (in fact arbitrary software calls) into a logical representation, which can be used in the planning process for *SHOP*. This allows us to formulate code call conditions: as statements in a logical language referring to arbitrary software functions.

To evaluate code call conditions (i.e., to determine whether they hold or not), we have to impose various restrictions on the underlying code calls. We have to ensure that code calls are

1. executable (i.e., return an answer),
2. only return finitely many answers.

Let us illustrate these problems with suitable examples.

Example 1 (Executable Code Call, Part 1) *We consider a software package* math *which provides several functions to handle integers (most agents will have available similar packages to do simple calculations). The code call* $\mathsf{math}: \mathit{geq}(\mathtt{X})$ *enumerates all integers greater or equal to* $\mathtt{X}$*.*

The code call $\mathsf{math}: \mathit{geq}(\mathtt{X})$ 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.

To ensure that code calls are always ground, we introduce the following safety condition:

Definition 14 (Safe Code Call (Condition)) *A code call* $\mathcal{S}: f(\mathtt{d_1}, \ldots, \mathtt{d_n})$ *is* safe *if and only if each* $\mathtt{d_i}$ *is ground. A code call condition* $\chi_1 \,\&\, \ldots \,\&\, \chi_n$*,* $n \geq 1$*, is* safe *if and only if there exists a permutation* π *of* $\chi_1, \ldots, \chi_n$ *such that for every* $i = 1, \ldots, n$ *the following holds:*

1. *If $\chi_{\pi(i)}$ is a comparison $\mathtt{s_1}$ op $\mathtt{s_2}$, then*

   1.1 *at least one of $\mathtt{s_1}, \mathtt{s_2}$ is a constant or a variable $\mathtt{X}$ such that $root(\mathtt{X})$ belongs to $RV_\pi(i) =_{def} \{root(\mathtt{Y}) \mid \exists j < i$ s.t. $\mathtt{Y}$ occurs in $\chi_{\pi(j)}\}$;*

   1.2 *if $\mathtt{s_i}$ is neither a constant nor a variable $\mathtt{X}$ such that $root(\mathtt{X}) \in RV_\pi(i)$, then $\mathtt{s_i}$ is a root variable.*

2. *If $\chi_{\pi(i)}$ is a code call atom of the form $\mathbf{in}(\mathtt{X_{\pi(i)}}, \mathtt{cc_{\pi(i)}})$ or $\mathbf{not\_in}(\mathtt{X_{\pi(i)}}, \mathtt{cc_{\pi(i)}})$, then the root of each variable $\mathtt{Y}$ occurring in $\mathtt{cc}_{\pi(i)}$ belongs to $RV_\pi(i)$, and either $\mathtt{X_{\pi(i)}}$ is a root variable, or $root(\mathtt{X_{\pi(i)}})$ is from $RV_\pi(i)$.*

Intuitively, a code call is safe if we can reorder the code call atoms occurring in it in such a way that we can evaluate these atoms left to right, assuming that root variables are incrementally bound to objects.

In [Eiter et al., 1999, Eiter and Subrahmanian, 1999, Subrahmanian et al., 2000] the authors developed algorithms to check safety for a given code call condition. Safety is a *compile-time* check that ensures that all code calls generated at *run-time* have instantiated parameters. However, executability of a code call condition does not depend solely on safety, as the next example shows.

Example 2 (Executable Code Call, Part 2) *We consider the code call*

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

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. This means that in general, we must impose some restrictions on code call conditions to ensure that they are finitely evaluable.

Indeed, it is well-known that deciding whether or not a function is finite is undecidable, therefore we assume that the developer of an agent examines the code calls supported by a given data structure and specifies which of them are finite and which are not.

Note that the following definitions 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.

Definition 15 (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 16 (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 17 (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 18 (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 [Subrahmanian et al., 2000].

We can now discuss the conditions for preserving soundness and completeness in *A-SHOP*. Consider the agentized method shown in Figure 4. It is reasonable to assume that the developer has defined FINTAB to entail code calls such as $\mathtt{supplier}:\mathit{cargoPlane}(\mathtt{locFrom})$. This means that the rows of FINTAB will have the form $\langle \mathtt{supplier}:\mathit{cargoPlane}(\mathtt{X}), (\flat)\rangle$, which says that calling any of these code calls with any values will succeed. Under these circumstances, it is easy to see that the method's precondition is strongly safe provided that the arguments of the method's head are instantiated. The reason is that the variable `CargoPL` is instantiated before `DCargoPl` and `CCargoPl`. Thus the code call is always ground.

We close this section by stating the conditions under which we can ensure the soundness and correctness of *SHOP* in *IMPACT*'s multi-agent environment. Our next theorem ensures the evaluation of agentized methods under the strongly safeness condition.

Theorem 3 *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.*

The next result is the main theorem in this paper: it states the correctness and completeness of *SHOP* if code-calls in methods and operators are strongly safe.

Theorem 4 *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.*

8 Implementation

The complete version of *SHOP* is built in LISP and includes the abilities to do Horn-clause inferencing and to make calls for 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 the `<URL:http://www.cs.umd.edu/projects/shop/>`

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 *Web-Hermes* [Adali, S., et al., 1997] 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 [Brink et al., 1995], a video information system called AVIS [Adali et al., 1996], 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 [Benton and Subrahmanian, 1994], a variety of US Army logistics data including specialized Oracle and nested multirecord TAADS data [Schafer et al., 1998], 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.

To facilitate the integration of *SHOP* in *IMPACT*, we re-implemented *SHOP* in JAVA. The java version of *SHOP* include 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 a theorem prover, *agentize* it using *IMPACT* methods and call the agent using an appropriate code-call condition ([Subrahmanian et al., 2000] 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, $\mathtt{math}$, is currently available in *IMPACT* to evaluate some numerical expressions.

We have built a communication module from *SHOP* to the *IMPACT* multi-agent environment that allows the execution of code-call conditions in *IMPACT* starting from *SHOP*. We are currently building a second communication module that communicates the results of the execution of *IMPACT*'s actions back into *SHOP*.

9 Discussion

Note that *A-SHOP* does not have any information about the state stored locally as it is usual in AI planning. 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 (1996) 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.

The use of *IMPACT*'s code call atoms differs from the built-in predicates of other approaches in that, first, code calls provide a well-defined semantics which allows us to state the conditions for preserving soundness and completeness of *SHOP*. Second, the integration of *SHOP* in the *IMPACT* environment allows us to address the challenges stated in the introduction:

- **Mixed symbolic/numeric reasoning.** Notice that the precondition of the method shown in Figure 4 supposes a combination symbolic and numeric reasoning. On the 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 information can be plugged in.

- **Distributed, heterogeneous information sources.** One important effect of integrating *A-SHOP* within *IMPACT* is that it allows 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 4 determining the statistics of a certain airplane may simply require access to a local database whereas determining if any such airplanes are available in a certain location may require access a remotely located spreadsheet. Recently, it has been observed that handling resources separate from the planning process may improve the performance [Srivastava and Kambhampati., 1999].

- **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 *A-SHOP* with 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.

10 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 *A-SHOP*, *SHOP*'s preconditions, add-lists and delete-lists are replaced with code call conditions. *IMPACT*'s code call conditions provide a well-defined syntax and most important they also provide a well-defined semantics. In particular, we have shown that *A-SHOP* is sound and complete provided that the code calls are strongly safe.

Knowledgments

We want to thank V. S. Subrahmanian who suggested using code call conditions in heterogeneous planning. We also want to thank Leonard A. Breslow and Peter Baumgartner for proof-reading early versions of this paper. This work was supported in part by the following grants and contracts: Army Research Laboratory DAAL01-97-K0135, Naval Research Laboratory N00173981G007, Air Force Research Laboratory F306029910013, and NSF DMI-9713718.

References

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

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

[Benton and Subrahmanian, 1994] Benton, J. and Subrahmanian, V. S. (1994). Using Hybrid Knowledge Bases for Missile Siting Problems. In Society, I. C., editor, *Proceedings of the Conference on Artificial Intelligence Applications*, pages 141–148.

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

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

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

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

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

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

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

[Erol et al., 1994] Erol, K., Hendler, J., and Nau, D. (1994). Umcp: A sound and complete procedure for hierarchical task-network planning. In *Proceedings of AIPS-94*.

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

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

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

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

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

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

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

[Nau et al., 1998] Nau, D. S., Smith, S. J. J., and Erol, K. (1998). Control Strategies in HTN Planning: Theory versus Practice. In *AAAI-98/IAAI-98 Proceedings*, pages 1127–1133.

[Schafer et al., 1998] Schafer, J., Rogers, T. J., and Marin, J. (1998). Networked Visualization of Heterogeneous US Army War Reserves Readiness Data. In Jajodia, S., Ozsu, T., and Dogac, A., editors, *Advances in Multimedia Information Systems, 4th International Workshop, MIS'98*, volume

1508 of *Lecture Notes in Computer Science*, pages 136–147, Istanbul, Turkey. Springer-Verlag.

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

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

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

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

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