ABSTRACT
There are many situations where an agent can perform one of several sets of actions in response to changes in its environment, and the agent chooses to perform the set of actions that optimizes some objective function. In past work, Eiter et al. have proposed a rule based framework for programming agents on top of heterogeneous data sources, but they provide no solutions to the above problem. In this paper, we propose a semantics called optimal feasible status set semantics for agents which allows the agent to associate an objective function with feasible status sets and act according to the feasible status set that optimizes this objective function. We provide both an algorithm to compute exact optimal feasible status sets as well as the TierOpt and FastOpt algorithms to find (sub-optimal) feasible status set much faster. We report on experiments on a suite of real agent applications showing that the heuristic algorithms works well in practice.

General Terms
Theory

Categories and Subject Descriptors
I.2.3 [Artificial Intelligence]: Deduction and Theorem Proving—Logic programming; I.2.11 [Artificial Intelligence]: Distributed Artificial Intelligence—Multiagent systems

Keywords
heterogeneous agents, non-ground representations, heuristic search, optimal status sets

1. INTRODUCTION
There is a long history of work on agents that maintain a current state and take one or more actions in response to a change in that state. More recently, Eiter et. al. [3] have proposed a framework for building agents on top of heterogeneous pieces of software code leveraging their scalable data structures as well as any application specific algorithms in that code. They propose a rule based language within which an agent developer can encode the operating principles (i.e. do’s and don’ts) of their agent. The semantics of these rules are expressed via a mathematical structure called a feasible status set (intuitively describing a set of actions the agent must perform in response to a state change). Agents can have zero, one or many feasible status sets. In such cases, Eiter et. al. [3] provide an optimal feasible status set semantics in which the agent chooses to act in accordance with one feasible status set (as opposed to another) using an objective function.

Such reasoning is very important in many applications. For example, consider a multiagent package delivery application. As circumstances change (e.g. blizzards, strikes, etc.), agents may have multiple ways of delivering packages to their destinations on time. The specific set of actions the agent may take (such as assigning packages to trucks, drivers to trucks, routes and manifest assignments to drivers, etc.) depend not only on what the agent is or is not allowed to do, but also on the objective function. The agent will want to choose a set of actions that satisfies the logical constraints on the behavior of the agent and optimizes the given objective function (e.g. minimize average delivery time, or minimize an expect dollar cost). An alternative application may be an intelligent airline sales agent which may automatically adjust prices for flight seats as the time of departure of the flight draws near. In such applications, the agent has various rules to honor (e.g. airline ticket pricing policies, pricing agreements with the government and corporate customers, etc.) while at the same time, optimizing an objective function such as expected profit on the route.

The optimal feasible status set (OFSS) semantics provided by Eiter et. al. [3] has two major problems.

1. First, Eiter et. al. define the semantics by assuming that agent rules must all be instantiated and that OFSSs are sets of ground status atoms (the formal definitions will be provided later). This has major performance implications. A single rule with 3 variables operating over a database system with 1000 constants in it has $10^9$ ground instances. Our first major contribution in this paper is a nonground optimal feasible status set semantics which is equivalent to the ground representation (but avoids unnecessary grounding).

2. Second, Eiter et. al. provide no algorithms to compute OFSSs. We first show that finding optimal status sets is NP-complete. We then develop an algorithm called OptFSS which is guaranteed to find an OFSS. As finding an OFSS is NP-complete, this algorithm is necessarily expensive. We develop two heuristic algorithms — TierOpt and FastOpt. Both these algorithms are much faster than OptFSS but may find a suboptimal solution.

3. Our third major contribution is an implementation of these algorithms. We describe experimental results comparing OptFSS, TierOpt and FastOpt. Our experiments compare the time taken
It is also important to note that code call conditions can span multiple software programs. For instance,
\[ \text{in}(M, \text{host : msgbox()}) \& \text{in}(M, \text{patient, P}) \& \text{in}(D, \text{oracle : match(doctors, specialty, X)}) \& \text{D.status = active}. \]

(cf. Figure 2) is a code call condition that spans both Oracle and the set computation source.²

An agent has access to a set of actions. An action is a piece of code written in some (most likely imperative) language. Every action has a name \( \alpha \) and a schema \((\tau_1, \ldots, \tau_n)\). Each \( \tau_i \) is a type with an associated domain \( \text{dom}(\tau_i) \). Associated with an action are a special code call condition called a precondition, and two distinct sets of code call conditions called the add-list and delete-list respectively, generalizing the standard AI definitions of actions. Intuitively, an action is executable in a given state if the precondition is true in that state. The result of executing the action is a new state just like the current state except that the code call conditions in the add list (resp. delete list) are true (resp. false).

An action atom is an expression of the form \( \alpha(t_1, \ldots, t_n) \) where \( \alpha(\tau_1, \ldots, \tau_n) \) is an action name and each \( t_i \) is either in \( \text{dom}(\tau_i) \) or is a variable ranging over \( \text{dom}(\tau_i) \). \( \text{assign}(D.name, P) \) is an action atom.

In general, the job of an agent is to determine what actions to take in response to a given state change. Agents use rules to determine how to react in the case of state changes. The rules are not necessarily hard and fast rules. For example, the rules may obligate the agents to perform certain actions in certain situations, but other actions may be permitted but not obligatory. In such cases, the agent can decide what actually to do based on any number of criteria (e.g. by using an objective function). We now give definitions to specify agent rules.

If \( \alpha \) is an action atom and \( OP \in \{P, O, O, F, W\} \), then \( OP \alpha \) is called a status atom.

For example, the status atom \( P\text{assign(doctor1, patient123)} \) may forbid allocating doctor1 to serve patient123. This is important as the doctor may have been forced to work continuously for 14 hours, making it illegal to allocate him further work without a rest period. Likewise \( P\text{am} \) means \( a \) is permitted, \( O\text{am} \) means \( a \) is obligatory, \( DO\text{am} \) means \( a \) is actually done, and \( W\text{am} \) means the obligation to do \( a \) is waived. These operators have a rich basis in deontic logic.

An agent rule is an expression of the form
\[ A \leftarrow \chi \land A_1 \land \ldots \land A_n, \]
where \( n \geq 0 \), \( A, A_1, \ldots, A_n \) are status atoms and \( \chi \) is a code call condition. All variables in a rule are universally quantified at the front of the rule. Such a rule, informally says “If the code call condition \( \chi \) is true in the current state and \( A_1, \ldots, A_n \) are all true, then \( A \) is true.” Figure 2 provides two sample rules.

**Note.** Throughout this paper, we assume, without loss of generality, that all rules only have variables in the heads. This leads to no loss of generality - for example, a rule such as \( DO\text{am}(X, a) \leftarrow Body \) can easily be rewritten as \( DO\text{am}(X, Y) \leftarrow Y = a \land Body \) where \( Y \) is a root variable that does not occur elsewhere in the rule.

### 2.1 Optimal Feasible Status Set Semantics

In this section, we describe the optimal feasible status set semantics proposed by Eiter et. al. [3].

A status set is a finite set of ground status atoms. It is very impor-

---

1 Virtually all commercial software come equipped with an API.

2 It is often convenient to think of an agent’s state as being synonymous with a set of ground (i.e. variable-free) code call conditions.
Suppose \( F \) is a feasible status set. It is a feasible status set for an agent \( a \) if and only if \( S \) is feasible w.r.t. \( a \) and \( \Omega \) if the following conditions are satisfied:

1. **Deontic consistency**: There is no ground action atom \( \alpha \) such that any of the following is true: \( \{ \text{DO}_a \alpha \} \subseteq S, \{ \text{DO}_a \alpha \} \subseteq S, \{ \text{DO}_a \alpha \} \subseteq S \). \( \{ \text{DO}_a \alpha \} \subseteq S \). \( \{ \text{DO}_a \alpha \} \subseteq S \). \( \{ \text{DO}_a \alpha \} \subseteq S \).

2. **Deontic closure**: \( S \) is closed under the following rules. (i) If \( \text{DO}_a \alpha \in S \) then \( \alpha \in S \). (ii) If \( \alpha \in S \), then \( \text{DO}_a \alpha \in S \).

3. **Action consistency**: If \( \alpha \in S \), then \( \text{pre}(\alpha) \) is true in \( \Omega \).

4. **Closure under program rules**: If \( \text{OP}_a \alpha_0 \leftarrow \chi \land \text{OP}_a \alpha_1 \land \cdots \land \text{OP}_a \alpha_n \) is a ground instance of a rule in \( a \)'s agent program and \( \chi \) is true in the current state \( \Omega \), then \( \text{OP}_a \alpha_0 \in S \).

5. **State consistency**: If we execute the set \( \{ \alpha | \text{DO}_a \alpha \in S \} \) in the current state \( \Omega \) to get a new state \( \Omega' \), then the new state \( \Omega' \) satisfies all integrity constraints associated with \( a \).

An agent may have zero, one, or many feasible status sets in a given state. A cost function \( \text{cost} \) takes a status set and state as input and returns a non-negative real number as output. For example, \( \text{cost}(\Omega, S) \) could measure the cost of performing the actions in \( S \) in state \( \Omega \), or it could measure the undesirability of the state that results by doing this, or it could be some linear combination of these two parameters.

**Example 2.** The following cost functions could be used for the agent shown in Figure 2.

- \( \text{cost}_1(S, \Omega) \) may return standard deviation in the number of patients assigned to a doctor in the state that results after executing the actions in \( S + 1000 \times \) number of patients assigned no doctor.

- \( \text{cost}_2(S, \Omega) \) may return the average number of hours on each doctor's shift in the state that results after executing the actions in \( S + 1000 \times \) number of patients assigned no doctor.

The above two cost functions heavily penalize status sets that allow patients to go without medical care. Of course, many other cost functions are also possible.

A status set \( S \) is said to be an **optimal feasible status set** for an agent \( a \) in state \( \Omega \) w.r.t. cost function \( \text{cost} \) iff

1. \( S \) is a feasible status set for agent \( a \) in state \( \Omega \), and
2. \( S \) is the standard deviation of the loads of the two doctors).

- \( \{ \text{assign}(doc1, pat1), \text{assign}(doc2, pat1) \} \) is feasible. However, its cost is \( \sqrt{5 + 1000} \) (\( \sqrt{5} \) is the standard deviation of the loads of the two doctors).

- \( \{ \text{assign}(doc1, pat1), \text{assign}(doc2, pat1), \text{doAssign}(doc1, pat1) \} \) is also feasible. Its cost is \( \sqrt{2} \).

- \( \{ \text{assign}(doc1, pat1), \text{assign}(doc2, pat1), \text{doAssign}(doc3, pat1) \} \) is also feasible. Its cost is \( \sqrt{5} \).

- \( \{ \text{assign}(doc1, pat1), \text{assign}(doc2, pat1) \} \) is also feasible. Its cost is \( \sqrt{5}/5 \).

Hence, the optimal status set would be the second one above which assigns doc1 to handle the patient.

The following result states that checking for optimality is an NP-hard problem.

**Theorem 1.** Suppose the cost() function is polynomially compatible. Suppose \( A \) is an agent in state \( \Omega \) and suppose \( S \) is a status set. The problem of determining if \( S \) is an optimal feasible status set is NP-hard.

### 3. Non-Ground Optimal Feasible Status Sets

Our goal is to compute feasible status sets that are optimal without grounding the agent program unless this is absolutely essential. This has a huge impact — in many of the example agents given in [5] on which this work is based, agents can have rules containing several variables. For example, the first rule shown in Figure 2 has four “root” variables \( D, P, M, X \). In a real hospital database, there may be (say 100 doctors for a small hospital), 1000 patients, a message box with 100 messages, and a list of 1000 symptoms. This would cause the first rule of Figure 2 to have a total of \( 10^6 \) ground instances (i.e. 100 million ground instances). An algorithm to compute optimal feasible status sets using an approach that first grounds out the rule would take a prohibitive amount of time to just ground the rules — this would render it useless. In this section, we provide an approach to avoid this by introducing “non-ground” computations. We first introduce “constrained conditions” and “constrained atoms.”

**Definition 1.** 1. Every code call condition is a constrained condition.

2. If \( \zeta_1, \zeta_2 \) are constrained conditions, then so are \( \zeta_1 \land \zeta_2 \) and \( \zeta_1 \land \zeta_2 \).

**Definition 2 (Constrained Status Atom).** If \( A \) is a status atom and \( \zeta \) is a constrained condition, then \( A \rightarrow \zeta \) is a constrained status atom.

A constrained status set is a finite set of constrained status atoms.

Intuitively, a constrained status atom is shorthand for all the “ground instances” of the head of the constrained status atom that are true in a given state - likewise, a constrained status set is also shorthand for all ground instances of all status atoms in it. This is more formally defined below.

**Definition 3.** Suppose \( A \rightarrow \zeta \) is a constrained status atom. The ground instance set of \( A \rightarrow \zeta \) in a state \( \Omega \), denoted \( \text{grd}_{\zeta}(A \rightarrow \zeta) \) is the set \( \{ A \theta | \theta \) is a substitution for variables in \( \zeta \) such that \( \emptyset \) is ground and \( \emptyset \) is true in state \( \Omega \) \}. When \( C \) is a constrained status set, we use \( \text{grd}_{\zeta}(C) \) to denote the set \( \bigcup_{A \in C} \text{grd}_{\zeta}(A) \).
Intuitively, $grd_2(A \leftarrow \zeta)$ denotes the set of all ground instances of $A$ that hold in the current state.

**Note.** We will often abuse notation and write $grd(A \leftarrow \zeta)$ instead of $grd_2(A \leftarrow \zeta)$ when $O$ is clear from context. In addition, in the rest of the paper, whenever we consider both agent programs and constrained status sets, we will always assume that different formulas are standardized apart (i.e. share no common variables). There is no loss of generality in making this assumption (we state it here so that we do not have to repeat this assumption repeatedly throughout the paper).

**Definition 4.** Suppose $C$ is a CSS. $C$ is an optimal CSS w.r.t. cost function $cost$ iff $grd(C)$ is an optimal feasible status set w.r.t. $cost$.

Intuitively, an optimal CSS is a nonground representation of an optimal feasible status set.

### 4. THE OptFSSAlgorithm

In this section, we define the OptFSS algorithm to compute an optimal feasible status set. In order to define the algorithm, several intermediate definitions are required.

**Definition 5.** Suppose $C$ is a CSS. $C$ is said to be deontically closed iff:

1. $DOA \leftarrow \zeta \in C \rightarrow PA \leftarrow \zeta \in C$ and
2. $A \leftarrow \zeta \in C \rightarrow DOA \leftarrow \zeta \in C \land PA \in C$.

We say that a CSS $C'$ is the deontic closure of $C$ iff:

1. $C'$ is deontically closed.
2. $\zeta \subseteq C'$.
3. If $C''$ is also deontically closed and $C \subseteq C''$, then $C' \subseteq C''$.

We use $DCl(C)$ to denote the closure of $C$.

It is easy to verify that the notion of deontic closure is well defined.

**Definition 6.** Suppose $P$ is an agent program. We associate with $P$ an operator $\Gamma_P$ that maps CSS to CSSs. $\Gamma_P(C) = C \cup \{OpA \leftarrow \chi \mid there is a rule in $P of the form $A' \leftarrow \chi' \land A_1 \land \ldots \land A_n$ and for each $1 \leq i \leq n, A_i' \leftarrow \chi_i \in C$ and }$A_i, A_i'$ are unifiable via a most general unifier $\theta$, and $A, A'$ are unifiable via a most general unifier $\theta$ and $\chi = \theta \land \bigwedge_i \{\theta_i, \chi_i\}$.

We use $\Lambda_P(C)$ to denote the set $DCl(\Gamma_P(C))$.

Intuitively, $\Gamma_P(C)$ returns $C$ together with all status atoms that can be derived from $C$ in one step using the rules in the agent program. Note that no grounding is required for this operator to work. $\Lambda_P$ “deontically closes” the result returned by $\Gamma_P$.

**Proposition 1.** $\Gamma_P(C)$ can be computed in cubic time w.r.t. the sizes of $P, C$.

The proof is based on the idea that for each element of $C$ and each rule in $P$, we merely need to check if it the head of the rule is unifiable with the selected element from $C$. Unification is linear. It is important to note that computing $\Gamma_P(C)$ is cubic in the sizes of $P, C$ and not w.r.t. the ground instantiation of either $P$ or $C$. This is a major improvement as the size of $grd_2(P)$ can be enormous.

Given a CSS $C$, we use the following notation to define the iterations of $\Gamma_P$ and $\Lambda_P$.

\[ \Lambda_P^0(C) = C \]
\[ \Lambda_P^{i+1}(C) = \Lambda_P(\Lambda_P^i(C)) \]
\[ \Lambda_P^\infty(C) = \bigcup_{i \geq 0} \Lambda_P^i(C) \]

The iterations of $\Gamma_P$ are defined in the same way, using $\Gamma_P$ in the above equations instead of $\Lambda_P$.

Given two CSSs $C_1, C_2$, we say that $C_1 \leq C_2$ iff $grd(C_1) \subseteq grd(C_2)$. The following theorem establishes some important properties of the $\Gamma_P$ and $\Lambda_P$ operators.

**Proposition 2.**

1. $\Gamma_P, \Lambda_P$ are both monotonic and continuous w.r.t. the $\leq$ ordering.

2. Therefore, $\Gamma_P$ has a least fixpoint denoted $lfp(\Gamma_P)$ and this least fixpoint is identical to $\Gamma_P^\infty(\emptyset)$.

3. For all $C, \Lambda_P^\infty(C)$ is the smallest fixpoint of $\Lambda_P$ which is greater than or equal to $C$ according to the $\leq$-ordering.

The following definition is used extensively in our OptFSS algorithm.

**Definition 7.** Suppose $OP_A, OP_A'$ are ground status atoms. We say that $OP_A, OP_A'$ conflict iff $A = A'$ and either:

1. $OP = P$ and $OP' = P$ or vice versa or
2. $OP = P$ and $OP = W$ or vice versa or
3. $OP = O$ and $OP' = P$ or vice versa or
4. $OP = DO$ and $OP' = F$ or vice versa or
5. $OP = DO$ and $OP' = W$ or vice versa.

Suppose $A$ and $B$ are CSS's. We say that $A$ and $B$ conflict if there exist $\theta$ and $\sigma$ such that, there are status atoms $A' \in grd(A\theta)$ and $B' \in grd(B\sigma)$ where $A'$ and $B'$ conflict.

**Definition 8.** Suppose $C$ is a CSS and $A$ is a ground set of status atoms. $C$ is expandable via $A$ if $\theta$ there is no $\theta'$ in $C$ such that $A$ and $A'$ conflict. We use $expand(C)$ to denote the set $\{C \cup \{A\} | C$ is expandable via $grd(A) \}$.

We are now ready to present the OptFSS algorithm.

**Algorithm 1.** function OptFSS($P, O$):

**Input:** $P$, an agent program; $O$, a state of the agent;

**Output:** an optimal feasible status set

1. done = false; bestSol = NIL; bestcost = $\infty$;
2. TODO = $\emptyset$; Done = $\emptyset$;
3. while TODO $\neq \emptyset$ do
4. $C = choice(TODO); (* choose any member of TODO *)$
5. $C' = \Lambda_P^\infty(C); Done = Done \cup \{\Lambda_P^\infty(C) \mid 0 \leq i \leq \infty\}$;
6. if $C'$ is deontically closed, action, and state consistent and
7. cost($C', O$) $< bestcost$ then
8. done = true; bestSol = $C'$;
9. best = cost($C', O$); bestSol = $C'$
10. else
11. TODO = $(TODO \rightarrow \{-C\}) \cup expand(C)$; Done = Done; return BestSol.

As we make no assumptions about the objective function, and as the problem of checking if a status set $S$ is an optimal feasible status set is NP-hard, OptFSS is bound to be slow. In general, if our agent has $n$ ground status atoms, and $card(\Lambda_P^\infty(\emptyset)) = n_0$ and $n_1 = n - n_0$, then the OptFSS algorithm takes time $O(n_0 + 2^{n_1})$. The following result states that OptFSS does indeed find an optimal feasible status set.

**Theorem 2.** (i) If $P$ has a feasible status set in state $O$ then OptFSS($P, O$) returns an optimal CSS.

(ii) If OptFSS($P, O$) returns $C$, then $C$ is an optimal CSS.

In view of the poor complexity properties of OptFSS, we now study fast algorithms that may compute a suboptimal solution.
5. **THE TierOpt ALGORITHM**

The basic idea in TierOpt is to start with $\Lambda_P(\emptyset)$. From [5], we know that any feasible status set must be equal to or is a superset of $\Lambda_P(\emptyset)$. The TierOpt algorithm uses parameters $m$ and $mch$ (which is a function to be described shortly) and a tree $T$. The root of $T$ is labelled with the CSS $\Lambda_P(\emptyset)$.

Suppose $n$ is a node in $T$ labelled with a CSS $S_n$. The function $mch(S_n)$ returns a set of $m$ ground status atoms none of which are in $grd_z(S_n)$, $mch(S_n)$ can be implemented in many different ways. For example, a choice function $mch_{1,\ldots,n}(S_n)$ may randomly pick $k$ ground atoms not in $S_n$. Alternatively, a choice function $mch_{k,\ldots,n}(S_n)$ may pick the $k$ ground atoms not in $grd_z(S_n)$ that each have least cost according to some algorithm for assessing cost of an atom. In our experiments, we will use $mch_3$, $cost$.

TierOpt starts by expanding the root by using the $m$ status atoms returned by $mch$. This leads to a total of $m$ children for the root. We expand each of these $m$ children as well, using $mch$ to get $m^2$ grandchildren of the root. We choose the best $m$ of the $(m + m^2)$ children and grandchildren of the root and delete the rest. The process continues in the same way. Whenever a child is generated, we check if it is feasible and if the objective function has a better value than the current best solution. If so, we update the best solution and best value and continue.

In the TierOpt algorithm given below, we use the function $expand_2(S, mch)$ to denote the set of all expansions of $S$ using an atom returned in the set $mch(S)$. When $X$ is a set of CSSs, we use select_CSS($X$) to denote a method to select some elements from $X$. There are many ways in which to pick select_CSS($X$). For example, we may decide that select_CSS($X$) will always return at most $r$ elements - we could choose these $r$ by giving some priorities to the various status atoms in $S$ that are linked to the cost of those status atoms.

**ALGORITHM 2. function TierOpt($P$, $O$, $m$, $mch$):**

**Input:** $P$, an agent program; $O$, a state of the agent.

**Output:** a (sub-)optimal feasible status set

1. $bestSet = \Lambda_P(\emptyset)$; $bestCost = \infty$;
2. $searchMore = true$
3. while ($searchMore$) do
4. $SA = bestSet$
5. $expandAtoms = mch(searchedAtoms)$
6. $SA = SA \cup expandAtoms$
7. $S = expand_2(bestSet, expandAtoms)$ // $|S| \leq m$; $|S| = 0$, stop.
8. if not foundBetter then
9. while (foundBetter)
10. if ($S = \emptyset$) $searchMore = false$; break while;
11. $X = (C_1, C_2, \ldots, C_m)$
12. $(E_1, E_2, \ldots, E_m) = (expand(C_1, mch(SA)), expand(C_2, mch(SA)), \ldots, expand(C_m, mch(SA)))$
13. $bestChild = chooseBest(E_1, E_2, \ldots, E_m, \{C_1, C_2, \ldots, C_m\})$
14. if (bestChild is feasible $\wedge bestChild.cost \leq bestCost$)
15. $bestSet = bestChild$
16. $bestCost = bestChild.cost$
17. foundBetter = true;
18. endif;
19. endwhile; foundBetter
20. endwhile; searchMore
21. return $bestSet$

6. **THE FastOpt ALGORITHM**

The FastOpt algorithm associates a cost, $cost(a)$ with each ground atom by setting $cost(a) = cost(DCL(a))$. As in the case of the TierOpt algorithm, it maintains a $bestSet$ variable showing the best CSS found so far and a $bestCost$ variable showing the cost of $bestSet$. Once $\Lambda_P(\emptyset)$ is computed, it iteratively adds the cheapest ground atom not already in $\Lambda_P(\emptyset)$ and closes it under the $\Lambda_P$ operator. If the result has a lower cost than the original, we replace the original with this set (greedy strategy), otherwise we keep both the original and the new generated set (branch). We maintain a tree structure where a new branch has as parent the set from which it split. If a set from the branch becomes better (in terms of cost) than the original root then we prune the tree by replacing the root with the best set from the branch. We call this algorithm the basic FastOpt algorithm.

In addition to this, we developed a variant of FastOpt called FASTes which includes an early stop strategy. In this strategy, we maintain a list of the sets generated in the last $N$ iterations of the algorithm. These sets are analyzed by a function called noProgress which determines if adequate progress is made (e.g. if $N = 4$ and the last four values obtained in the iteration are $100, 99, 99, 99$ then we may decide to stop the iterations). noProgress can be implemented in any number of ways - one way is to look at the maximum and minimum values for the last $N$ iterations and terminate if the difference between them is less than some threshold value. Another method to implement noProgress is to use a statistical model where the standard deviation is examined over a time window — if the standard deviation is below some small threshold, then we assume that inadequate progress is being made and terminate FASTes. FastOpt’s code can be obtained from the code below by dropping the pruning that occurs using noProgress.

**ALGORITHM 3. Input:** an agent with agent program $P$, and $O$: a state of the agent;

**Output:** estimate of the optimal feasible status set

1. $initSet = \Lambda_P(\emptyset)$
2. $bestSet = initSet$
3. $bestCost = cost(initSet, O)$
4. $searchList = initSet$
5. $stopList = \emptyset$
6. $sortedAtoms = sortGroundStatusAtoms();$
7. while ($sortedAtoms <> \emptyset$ and $cost(nextAtom) < 0$) do
8. for all CSS in $searchList$
9. $CSS = \Lambda_P(\emptyset) \cup \{nextAtom\}$
10. if ($cost(CSS, O) < cost(CSS, O)$) then
11. $searchList = searchList - \{CSS\} \cup \{CSS\'}$
12. else $searchList = searchList \cup CSS'$
13. $if (cost(CSS', O) < bestCost)$
14. $bestCost = cost(CSS', O)$
15. $bestSet = CSS'$
16. $endif;
17. $stopList = stopList \cup CSS'$
18. $endif$
19. $if noProgress(stopList)$
20. $break$
21. $prune(searchList)$
22. $endwhile
23. return $bestSet$

7. **IMPLEMENTATION AND EXPERIMENTS**

We developed about 10,000 lines of Java code implementing the above algorithms on top of the IMPACT system [5] for building agents on top of legacy code. We tested our approach using four agents built by the authors of IMPACT which they provided us. All these agents were built on top of Microsoft Access. The “Tax” agent is a variant of an agent in [5] that sends notifications to taxpayers. Deploy is an agent that computes a deployment strategy for some troops in a battle field. Register is an agent that allows users

\footnote{Thanks to the authors of IMPACT for making their software available.}
FAST es

**Optimal Answer**

the three algorithms (Experiment 1: each algorithm to return and print an answer.

Flight reservation is an agent when the first request arrives, and becomes profitable only if a certain number of users register for it. The service has to be initialized to register with a service provider. The service used was small owing to the fact that developers of IMPACT[5].

Similar patterns were observed in other experiments we conducted as well. Overall, we can see that FAST es performs comparably in terms of the quality of the answer found, and performs much faster than TierOpt in terms of the computation time. We therefore concluded that FAST es is the better algorithm.

### 8. RELATED WORK AND CONCLUSIONS

Our work builds directly on top of important work in logic programming on developing a nonground semantics for logic programs. In particular, we build on work by Turi [6] who took positive logic programs and designed an equality/inequality constraint based fixed point semantics for it. Later, Eiter et al.[2] extended the approach of Turi[6] to handle negation as well. Many other extensions of this basic semantics have also been developed [1, 4].

Though our work builds directly upon these methods, it is different in four important respects. First, our notion of using code conditions to build agents on top was not considered in preceding works. Second, previous works did not account for the deontic modalities we have in our work, or in capturing the feasible status set semantics. Third, none of these works developed algorithms for optimal status set computations. Fourth, none of these works implemented any algorithms. To our knowledge this is the first work on non ground computation of optimal feasible status sets.

Much future work remains to be done. For example, are there conditions under which optimal feasible status set computation is polynomial? Are there versions of optimal status sets that apply to other kinds of status sets such as rational and reasonable status sets defined in [5].

### Acknowledgements

Work supported in part by ARO grant DAAD190310202, ARL grants DAAD190320026 and DAAAL0197K0135, the ARL CTAs on Telecommunications and Advanced Decision Architectures, and NSF grants IIS0329851 and 0205489.

### 9. REFERENCES


<table>
<thead>
<tr>
<th>TierOpt</th>
<th>FAST es</th>
<th>Optimal Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>-219</td>
<td>-162</td>
<td>-219</td>
</tr>
<tr>
<td>-244</td>
<td>-244</td>
<td>-244</td>
</tr>
<tr>
<td>-161</td>
<td>-161</td>
<td>-161</td>
</tr>
<tr>
<td>-235</td>
<td>-235</td>
<td>-235</td>
</tr>
<tr>
<td>-189</td>
<td>-172</td>
<td>-189</td>
</tr>
<tr>
<td>-148</td>
<td>-148</td>
<td>-148</td>
</tr>
<tr>
<td>-219</td>
<td>-219</td>
<td>-219</td>
</tr>
<tr>
<td>-108</td>
<td>-108</td>
<td>-108</td>
</tr>
<tr>
<td>-232</td>
<td>-205</td>
<td>-232</td>
</tr>
<tr>
<td>-198</td>
<td>-196</td>
<td>-198</td>
</tr>
</tbody>
</table>

**Table 1: Answer quality: Deploy Agent**

<table>
<thead>
<tr>
<th>TierOpt</th>
<th>FAST es</th>
<th>Optimal Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>-32</td>
<td>-32</td>
<td>-32</td>
</tr>
<tr>
<td>-43</td>
<td>-43</td>
<td>-43</td>
</tr>
<tr>
<td>0</td>
<td>-22</td>
<td>-22</td>
</tr>
<tr>
<td>0</td>
<td>-32</td>
<td>-32</td>
</tr>
<tr>
<td>0</td>
<td>-69</td>
<td>-69</td>
</tr>
<tr>
<td>57</td>
<td>-57</td>
<td>-57</td>
</tr>
<tr>
<td>56</td>
<td>-56</td>
<td>-56</td>
</tr>
<tr>
<td>42</td>
<td>-42</td>
<td>-42</td>
</tr>
<tr>
<td>0</td>
<td>-44</td>
<td>-44</td>
</tr>
<tr>
<td>21</td>
<td>-21</td>
<td>-21</td>
</tr>
</tbody>
</table>

**Table 2: Answer quality: Register Agent**

<table>
<thead>
<tr>
<th>TierOpt</th>
<th>FAST es</th>
<th>Optimal Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>-432</td>
<td>-432</td>
<td>-513</td>
</tr>
<tr>
<td>-489</td>
<td>-489</td>
<td>-538</td>
</tr>
<tr>
<td>-512</td>
<td>-512</td>
<td>-512</td>
</tr>
<tr>
<td>-550</td>
<td>-550</td>
<td>-602</td>
</tr>
<tr>
<td>-459</td>
<td>-459</td>
<td>-506</td>
</tr>
<tr>
<td>-537</td>
<td>-537</td>
<td>-555</td>
</tr>
<tr>
<td>-474</td>
<td>-474</td>
<td>-557</td>
</tr>
<tr>
<td>-498</td>
<td>-498</td>
<td>-526</td>
</tr>
<tr>
<td>-580</td>
<td>-580</td>
<td>-680</td>
</tr>
<tr>
<td>-525</td>
<td>-525</td>
<td>-569</td>
</tr>
</tbody>
</table>

**Table 3: Answer quality: Flight Agent**

found a much better solution than TierOpt. This was observed consistently across varying numbers of ground atoms. Likewise, in the case of the Deploy agent, we found that FAST es takes much less time than TierOpt (though the difference was less pronounced than in the case of the Taxes agent). In this case, TierOpt actually found better solutions in some cases.

As the reader can see, in all three cases, these algorithms come very close to the optimal solution found using OptFSS. In addition, there is very little to distinguish between the solution found by TierOpt and by FAST es. Both find very good solutions and there are cases where each finds better solutions than the other.

**Experiment 2:** Experiment 1 did not assess the relative efficiency of FAST es versus TierOpt. In our second experiment, we compared how these two algorithms perform, both in terms of quality of the answer found, and in terms of the computation time involved.

As the reader can see, the FAST es algorithm wins by a huge order of magnitude in terms of the time taken. For example, with a total of 5950 ground atoms, FAST es only takes 16 seconds, compared with 380 for the TierOpt algorithm. In addition, it actually to register with a service provider. The service has to be initialized when the first request arrives, and becomes profitable only if a certain number of users register for it. Flight reservation is an agent that books flight tickets and attempts to maximize expected profit for the seller. Times reported below include the total time taken for each algorithm to return and print an answer.

**Experiment 1:** We studied the quality of the answers provided by the three algorithms (OptFSS, FAST es, TierOpt) on the Deploy, Register and Flight agents in the form provided to us by the developers of IMPACT[5]. The data used was small owing to the fact that OptFSS (as expected) takes an inordinate amount of time. Scalability will be studied in the next experiment. Tables 1, 2, and 3 show the results. The numbers shown denote the values of the objective functions used. All the problems required maximizing an objective function — we minimized the negative of the objective function (which is equivalent. This is why the numbers shown in the table are negative.

As the reader can see, in all three cases, these algorithms come very close to the optimal solution found using OptFSS. In addition, there is very little to distinguish between the solution found by TierOpt and by FAST es. Both find very good solutions and there are cases where each finds better solutions than the other.

**Experiment 2:** Experiment 1 did not assess the relative efficiency of FAST es versus TierOpt. In our second experiment, we compared how these two algorithms perform, both in terms of quality of the answer found, and in terms of the computation time involved.

As the reader can see, the FAST es algorithm wins by a huge order of magnitude in terms of the time taken. For example, with a total of 5950 ground atoms, FAST es only takes 16 seconds, compared with 380 for the TierOpt algorithm. In addition, it actually...
Table: Performance and Answer Quality

<table>
<thead>
<tr>
<th>no. of ground atoms</th>
<th>TierOpt Search Time(s)</th>
<th>TierOpt Search Value</th>
<th>FASTes Search Time (s)</th>
<th>FASTes Search Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>0.9</td>
<td>-100</td>
<td>0.03</td>
<td>-100</td>
</tr>
<tr>
<td>1900</td>
<td>47</td>
<td>-2120</td>
<td>1.2</td>
<td>-3800</td>
</tr>
<tr>
<td>3510</td>
<td>129</td>
<td>-4640</td>
<td>5</td>
<td>-7020</td>
</tr>
<tr>
<td>5950</td>
<td>380</td>
<td>-7860</td>
<td>16</td>
<td>-11900</td>
</tr>
</tbody>
</table>

Figure 2: Performance and answer quality: Taxes Agent

Figure 3: Performance and Answer quality: Deploy Agent


