Continuous Refinement of Agent Resource Estimates

David N. Morley Karen L. Myers Neil Yorke-Smith

Artificial Intelligence Center, SRI International
Menlo Park, CA 94025 U.S.A.
{morley,myers,nysmith}@ai.sri.com

ABSTRACT

The challenge we address is to reason about projected resource usage within a hierarchical task execution framework in order to improve agent effectiveness. Specifically, we seek to define and maintain maximally informative guaranteed bounds on projected resource requirements, in order to enable an agent to take full advantage of available resources while avoiding problems of resource conflict. Our approach is grounded in well-understood techniques for resource projection over possible paths through the plan space of an agent, but introduces three technical innovations. The first is the use of multi-fidelity models of projected resource requirements that provide increasingly more accurate projections as additional information becomes available. The second is execution-time refinement of initial bounds through pruning possible execution paths and variable domains based on the current world and execution state. The third is exploitation of additional semantic information about tasks that enables improved bounds on resource consumption. In contrast to earlier work in this area, we consider an expressive procedure language that includes complex control constructs and parameterized tasks. The approach has been implemented in the SPARK agent system and is being used to improve the performance of an operational intelligent assistant application.

Categories and Subject Descriptors

I.2.8 [**ARTIFICIAL INTELLIGENCE**]: Problem Solving, Control Methods, and Search—*Plan execution, formation, generation*

General Terms

Algorithms

Keywords

Belief-Desire-Intention agents, resource approximation, plan execution, procedural knowledge

1. INTRODUCTION

The challenge we address is to provide reasoning about projected resource usage within a hierarchical task execution framework to enable improved agent effectiveness. In particular, we seek to provide *guaranteed* bounds on resource consumption consisting of a lower bound that describes necessary resource requirements and an upper bound that describes possible resource requirements. The range between these bounds can stem from uncertainty about the current world, uncertainty about future actions and events, and ambiguity about the consumption requirements of actions.

Guaranteed resource bounds are essential for informing an agent's deliberations for task adoption and strategy selection because they enable it to make *conservative* commitments that eliminate the possibility of resource contention [15, 3]. With such bounds, an agent can avoid adopting tasks or performing actions that are resource infeasible given the agent's other commitments. In addition to being guaranteed, resource bounds should be *maximally informative*, meaning that they incorporate as much information as is available, in as timely a fashion as is possible. This requirement introduces the need to provide execution-time updates of bounds in response to both runtime events and the acquisition of relevant knowledge. By updating the bounds during execution, an agent can reassess its task set [11], for instance, to suspend or drop a task that consumes more of a resource than anticipated, or to adopt a task that becomes resource feasible.

Current techniques for agent resource projection are inadequate. Purely reactive agents perform no lookahead, acquiring knowledge about resource usage only as it occurs. Problems with resource contention are handled as they arise, which can be both costly and failure prone. Methods from the AI planning literature show how to derive guaranteed initial bounds on resource consumption [6, 2], but they require that estimates for all resource-consuming tasks be available before execution commences. Furthermore, they do not address the issue of updating resource projections at execution time. Previous work that combines lookahead and reactive techniques uses overly simplistic task models, reasoning about resource usage only for propositional tasks (i.e., not parameterized by variables) without branching or iteration [16].

Our approach is a middle ground focused on limited projection and runtime updates, in the spirit of [16]. The contributions are threefold. First, we adopt a *multi-fidelity* model of resource production and consumption that supports estimates whose accuracy can vary with the information available about tasks. One of the difficulties in estimating resource requirements accurately is that future tasks may depend on parameters that have not yet been instantiated. Our multi-fidelity models are valuable when bounds change significantly depending on how individual parameters are instantiated. For example, consider a task $(\mathsf{Flight}\ \langle\text{from}\rangle\langle\text{to}\rangle\langle\text{airline}\rangle\langle\text{class}\rangle)$. Without any knowledge of the starting point and destination, the bound on resource *money* would be very loose. If the $\langle\text{from}\rangle$ and $\langle\text{to}\rangle$ parameters are known, then the bound may be reduced substan-

AAMAS'06 May 8–12 2006, Hakodate, Hokkaido, Japan.

tially; similarly, the choice of airline could limit fares (e.g., budget vs. full-service airlines), as could the class (e.g., first vs. coach). Our multi-fidelity models allow different bounds to be specified for each of these variations, at the discretion of the domain modeler.

Second, we refine our initial resource projections as execution unfolds. These updates are performed at the end of the observation phase in the observe–decide–act cycle of the agent. They include not only pruning possible execution paths based on the current world and execution state, but also reasoning over and narrowing variable domains. Thus we ensure that information about resources derivable from execution-time events is made available to the agent's deliberation at the earliest possible stage. Third, we show how certain kinds of additional semantic information about tasks can be exploited to improve resource bounds.

Our work is grounded in an expressive procedure language that can be viewed as an abstraction of two sophisticated procedural reasoning frameworks for Belief-Desire-Intention (BDI) agents [13], namely, PRS [7] and SPARK [9]. The language supports a range of parameterized task types, including condition achievement, task performance, and waiting for or testing a condition. Complex procedures can be composed from these task types through the control constructs of sequencing, parallelism, conditional branching, iteration, and subgoaling.

Our resource projection methods have been realized in the SPARK system and applied to improve the effectiveness of resource usage within an intelligent personal assistant domain [14].

2. RESOURCES: MODELS AND USAGE

Resource projection requires models of production and consumption for the tasks that can be performed within a domain. In this section, we present a model for *deterministic* resource usage; the nondeterministic case is left for future work. An important distinction is between consumption of *replaceable* resources, which are returned by a task after it completes, and *non-replaceable* resources. For example, a conference room or a transferable software license is a replaceable resource, while money is non-replaceable. However, money is a resource that can be *produced*. We assume a simple temporal model in which a task consumes resources when it commences, and releases (or produces) resources when it (successfully) completes. We do not consider scheduling of tasks in this paper. Resources may be discrete or continuous, and sharable between tasks or not.

In certain cases, resource requirements can be modeled precisely in advance: when the amount of resource consumed or produced is either a constant (e.g., $15 to download an article from ACM, $100 to change an airline reservation), or a parameterized function of the task's parameters (e.g., (RegisterForConference$\langle X \rangle$) — fees vary across conferences but are known once registration opens). We call resource requirements that can be modeled precisely in advance *knowledge pre facto*. These resource requirements can be modeled as an n-ary function of some subset of the task's parameters, with $n = 0$ in the case of a constant resource requirement.

In other cases, resource consumption or production for a task may not be predictable in advance (e.g., the cost of an airline ticket). In such cases, only loose bounds on the resource requirements may be obtained unless explicit actions are taken to determine the requirements (e.g., by checking the cost online). Further, certain types of resource consumption cannot be measured precisely until the action completes (e.g., how much petrol to drive to the airport); we label such requirements *knowledge ad facto*. For other tasks, there may be some delay in determining the precise consumption level (e.g., posting of a charge in foreign funds to your credit card); we label such requirements *knowledge post facto*. Similarly, certain types of resource production may also be knowledge post facto.

We say that a task is *knowledge pre facto* (similarly, *ad* or *post facto*) if it is so for all its resources. In this paper we avoid the complications of knowledge post facto tasks by assuming that all consumption and production are knowledge pre or ad facto.

In our approach to modeling resource requirements, each task that directly consumes (produces) a resource includes an explicit effect for that consumption (production). A *resource model* consists of a set of *approximations* that provide strict lower and upper bounds on the amount of a particular resource R required for (produced by) a given task:

$$(\ \langle R \rangle\ \langle \mathit{appr}_0 \rangle \langle \mathit{appr}_1 \rangle \ldots \langle \mathit{appr}_m \rangle\ ) \tag{1}$$

The approximations $\langle \mathit{appr}_i \rangle$ are specified in terms of functions of some subset of the task parameters that specify lower and upper bounds on the resource consumption (production):

$$[\underline{f_i}(p_1, \ldots, p_{s_i}),\ \overline{f}_i(q_1, \ldots, q_{t_i})] \tag{2}$$

In cases where $\underline{f_i} = \overline{f}_i$, we say that the bound is *precise*. For simplicity of presentation, we assume that the functions used to compute lower and upper bounds share the same set of arguments (i.e., in (2) above, $s_i = t_i$ and $\forall j.\ p_j = q_j$). Projected resource estimates, as explained later, should take into account all possible approximations as they become computable, that is, as all the parameters to the approximation functions have been instantiated.

A typical usage of the construct (1) would be something like the following for a resource *money* consumed by an ArrangeCateringExternal task: $(\mathit{money}\ [(\times\ \$\mathsf{n}\ 10),\ (\times\ \$\mathsf{n}\ 20)]\ \ [120,\ 1000])$, where $\mathsf{n} is the number of people to be catered.

3. RESOURCE PROJECTION

3.1 Overview

Our resource projection framework provides *guaranteed* bounds on the use of resources for execution of a task, where the lower bound describes *necessary* and the upper bound describes *possible* resource requirements. Given the requirement of guaranteed bounds, it is sound for the projection to derive over-estimates of these bounds. Of course, excessively broad bounds may be of little or no value to an agent when deliberating over task feasibility.

Set against this background, our approach to resource projection is one of *limited* static lookahead prior to task execution, followed by dynamic updates during execution. The static projection considers the space of possible decompositions of the task, drawing on the set of plans available to the agent. The execution-time updates refine the static projections by incorporating information about execution decisions and changes to dynamic world state.

Motivated by the tasks and processes found in the personal assistant domain [11, 14], which are dominated by the use of resources to achieve tasks delegated by the user, we focus on resource consumption. Combining production and consumption is important but beyond the scope of this paper.

In the personal assistant domain, as in many others, failures can be classified as *minor* or *major*. Whereas many minor failures can be remedied by increasing resource consumption (e.g., the flight cost more than expected), major failure entails reconsideration of the task (e.g., there are no flights available and the meeting will need to be rescheduled).

Our resource projection framework does not explicitly account for task failure, focusing instead on resource needs for failure-free execution. Our expectation is that an agent that makes use of this framework would maintain a resource reserve from which it would

```
{defprocedure plan_group_visit
  cue:  [do: (planGroupVisit $visitors)]
  body: [sequence:
    [parallel:
      [do: (getPossibleDates $visitors $dates)]
      [do: (solicitInterested $visitors $people)]]
    [do: (scheduleDay $dates $visitors $people
                      $schedule)]
    [select:
       (ClearanceRequired $visitors)
          [sequence:
             [conclude: (InternalOnly $schedule)]
             [do: (applyForClearance $visitors)]]
       (True)  % otherwise
          [retract: (InternalOnly $schedule)]]
    [context: (Concat $visitors $people $all)
      parallel:
        [do: (informSchedule $all $schedule)]
        [do: (publicizeSeminar $schedule)]
        [do: (arrangeCatering $all $schedule)]]]}

{defprocedure apply_for_clearance
  cue: [do: (applyForClearance $visitors)]
  body: ...
  consumes: [(money (+ 100 (* 50 (length $visitors)))
                    [100 500])
             (hours 1)]
  requires: [(licenses 1)]}

{defprocedure publicize_seminar
  cue:  [do: (publicizeSeminar $schedule)]
  body: [sequence:
    [do: (publicizeSeminarInternal $schedule)]
    [do: (getPublicityCostEstimate $schedule $cost)]
    [select: (and ($cost < 50)
                  (not (InternalOnly $schedule)))
                [do: (publicizeSeminarExternal $schedule)]
             (True)  % otherwise
                [succeed:]]]}

{defprocedure arrange_catering_external
  cue:  [do: (arrangeCatering $everyone $schedule)]
  body: ...
  consumes: [(money [(* (length $everyone) 10)
                     (* (length $everyone) 20)]
                    [120 1000])
             (hours [1 3])]
  requires: [(licenses 1)]}
```

Figure 1: Plans for the group visit task

draw to remedy minor failures. For a major failure, a BDI agent will reconsider its commitments and decide upon a new course of action [13], which will in turn lead to the recomputation of projected resource bounds. For this reason, our projected bounds do not include resource requirements for addressing failures.

Our approach for projecting guaranteed resource bounds for a task presumes a task and plan model described in Sect. 3.2. The first step in the projection process is to generate a *task expansion tree* that encodes the space of possible decompositions for the task with respect to the agent's plan library (Sect. 3.3). For that structure, a *variable relationship graph* is then constructed that encodes relationships and constraints among parameters of the considered plans; constraint programming techniques are used on this structure to bound possible values of task parameters (Sect. 4). The resource projection itself involves determining resource requirements for leaf nodes in the task expansion tree, drawing on information from the variable relationship graph as to possible or actual bindings for task parameters, and then aggregating those values up through the task expansion tree to produce the overall resource summary for the root task (Sect. 3.4).

3.2 Tasks and Plans

To make concrete our description, we introduce an explicit task and plan model based loosely on the SPARK framework [9].

A *task* is a parameterized activity for the agent to execute. Tasks come in two types. *Simple* tasks (`achieve`, `do`, `wait`, `conclude`, `retract`, `succeed`, `fail`) are the more basic form of task. Of these, only `achieve` and `do` can consume resources. An `achieve` task is used to achieve a designated state condition, while a `do` task is used to perform a designated action. A simple task can be *primitive*, meaning that the agent's effectors can execute it directly, or *nonprimitive*. A nonprimitive task is executed by decomposing it into lower-level subtasks.

Composite tasks (`sequence`, `parallel`, `select`, `context`, `try`, `forall`, `while`) can be used to compose aggregate tasks from a set of *component* tasks in a functional manner. For example, the composite task [`sequence`: $T_1, \ldots, T_m$] denotes the sequential performance of the component tasks $T_1, \ldots, T_m$. The task type `select` provides for the choice among component tasks, conditioned on the outcome of a sequence of test conditions; `try` similarly supports branching, but conditioned on the successful performance of some test task; `context` provides for the testing of predicates for the purpose of binding variables. The semantics for the remaining task types follow standard interpretations (see [9]).

A *plan*, referred to as a *procedure* in SPARK terminology, describes one of possibly many ways of decomposing a nonprimitive task. (SPARK plans can also be used to respond to a change in the agent's beliefs — a capability not considered in this paper.) Each plan has a *cue* that indicates the event that triggers the plan, an optional *precondition* that states the conditions under which the plan is applicable, and a *body* that consists of a single task (either simple or composite) to be performed in response to the plan cue.

As noted above, resource models are associated with tasks. Given the definition of a plan body as a (possibly composite) task, we can similarly associate a resource model with a plan.

EXAMPLE 1. *Figure 1 shows extracts of plans for a group visit task in this language. Variables are prefixed by* `$`*. Resource approximations annotate the plans* `apply_for_clearance` *and* `arrange_catering_external`*. For instance, the latter has a resource approximation corresponding to* (*money* $[10n,\ 20n]$ $[120,\ 1000]$)*, where* n *is the length of the list* `$everyone`*. In this example, the resources money and hours are nonreplaceable (denoted by* `consumes` *in the resource approximation), while licenses are replaceable (denoted by* `requires`*).*

3.3 Task Expansion Tree

The *task expansion (TE) tree* is the principal data structure used for resource projection. Intuitively, a TE tree for a task T describes all possible resource-related (producing and consuming) subtasks that could be performed to accomplish T. It is obtained by considering how T can be expanded into subtasks via application of plans in the agent's plan library.

The root node of a TE tree corresponds to the task T. A TE tree contains both simple and composite task nodes. Each simple task node has a child node corresponding to every possible matching plan that could achieve the task; the child node denotes the task (either simple or composite) that defines the body of the associated plan. Thus, a simple task node has *or* semantics in that its children represent alternative means of achieving the task. We call the edge linking a simple task to one of its children a *plan edge*. Each composite task node has one or more child nodes corresponding to the component tasks for the composite task; links from a composite task node to its children are called *structural edges*.

The TE tree for a given task is constructed recursively. The child nodes of a simple task node are obtained by matching the possible plans, according to the task name and parameters (as far as they are instantiated), and current constraints on possible matchings. This matching considers static predicates only. The child nodes of a composite task node are obtained by decomposing the task into its component tasks. Construction halts on a given branch when the child node is either a primitive task or a task with a resource approximation. (Note that resource models can be specified for both primitive and nonprimitive tasks, at the discretion of the modeler.)

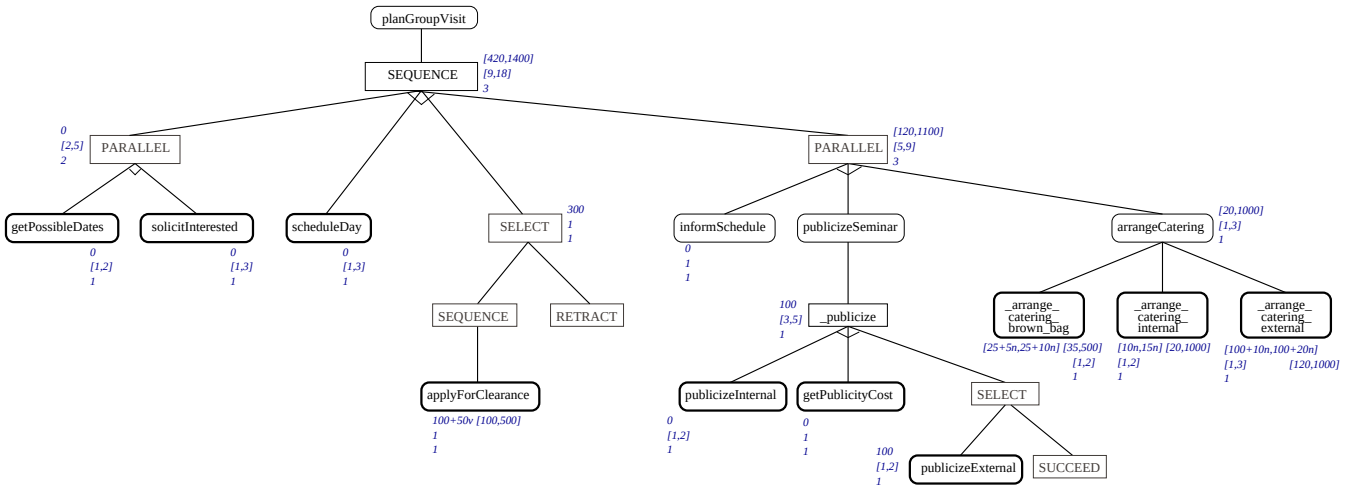

Figure 2: Task expansion tree for the task `planGroupVisit`

With the assumption of a nonrecursive library of plans, the tree and its construction are guaranteed to be finite.

Every node in a TE tree has a *resource summary* that provides for each resource the aggregate necessary and possible bounds on the consumption levels for that node and its descendants:

$$\langle\ (R_1\,[L_1,\,U_1]),\ \ldots,\ (R_m\,[L_m,\,U_m])\ \rangle \tag{3}$$

Here, L_i and U_i represent guaranteed lower and upper bounds, respectively, on the use of resource R_i for failure-free execution.

EXAMPLE 2. *The TE tree for the task* `planGroupVisit` *is displayed in Figure 2. Simple task nodes are depicted with rounded boxes and composite nodes with square boxes; bold indicates a task with a resource approximation model annotated. For simplicity, neither the parameters of tasks nor any* `conclude` *tasks are shown. Adjacent to each node is its resource approximation (for leaf nodes) or resource summary (for non-leaf nodes). These are shown as three lines for the resources money, hours, and licenses, respectively. Here,* n *represents the length of the list* `$everyone` *(recall Example 1), and* v *is the number of visitors, which is an input parameter to the root task.*

3.4 Computation of Resource Summaries

The computation of resource summaries for a TE tree involves first computing resource bounds for individual leaf nodes (Sect. 3.4.1), and then aggregating the bounds up to the root (Sect. 3.4.2). We assume that the domain model includes a *comprehensive* set of resource approximations for the domain, meaning that each primitive task that consumes resources has an associated resource approximation providing a guaranteed bound on its resource requirements.

3.4.1 Resource Bounds for Leaf Nodes

The resource summary for a leaf node T in a TE tree is determined by considering all resource approximations for the task of T (of the form (1)) whose parameters are fully instantiated. Let the bounds produced by this set of approximation functions be $\{[L_1,\,U_1],\ldots,[L_m,\,U_m]\}$. For a primitive task that does not consume resources (as reflected by the lack of a resource approximation in the domain model), this set is defined to be $\{[0,\,0]\}$. These bounds are strict, but represent different perspectives on the range of possible requirements; as such, they must be intersected as follows to produce the maximally refined bound for T:

$$\prod_{i=1,\ldots,m} [L_i,\,U_i] = [\max_{i\in\{1,\ldots,m\}} L_i,\ \min_{i\in\{1,\ldots,m\}} U_i] \tag{4}$$

It is not always necessary that a parameter to a resource approximation $[\underline{f},\,\overline{f}]$ be bound in order for the approximation to be used. Valid (but perhaps looser) bounds can be computed by inference over the variable's current domain, using techniques from constraint programming [1, 5]. In particular, in situations where the domain for the variable is restricted to a finite enumerable set $D = \{d_1,\ldots,d_k\}$, reasoning by cases can be applied to D yielding the bounds $L_i = \min_{d\in D} \underline{f}(d)$ and $U_i = \max_{d\in D} \overline{f}(d)$ on the individual approximations. The same enumeration technique applies to the Cartesian product of the domains of a set of finite domain variables, although care must be taken to consider only domains of tractable size.

3.4.2 Resource Aggregation for Non-Leaf Nodes

Resource aggregation computes the resource summary (3) for a non-leaf node T in a TE tree based on the summaries of its children. We assume no constraints between resources, that is, each resource is consumed and produced independently of others. In this case, it suffices to consider each resource in turn.

Consider first a simple task node T with child nodes $T_1,\ldots,T_m$ where T_i has resource summary $[L_i,\,U_i]$ for a given resource. As noted above, our model presumes that of the simple tasks, only `achieve` and `do` can impact resource usage. Recall that each child of a simple task node corresponds to the body of a possible plan for achieving the task. At runtime, exactly one of the T_i will be chosen for execution. Thus the resource summary for T consists of a lower bound that is the minimum of the L_i and an upper bound that is the maximum of the U_i, i.e., $[\min L_i,\,\max U_i]$.

Now consider a composite task node T with component task nodes $T_1,\ldots,T_m$; again let the child nodes have summaries $[L_i,\,U_i]$. Replaceable and non-replaceable resources differ in their aggregation. Adapting [16], we define the composition operator:

$$x \otimes y = \begin{cases} \max(x,y) & \text{for replaceable resources} \\ x+y & \text{for non-replaceable resources} \end{cases}$$

Aggregation for different types of composite nodes is defined as follows. We do not consider `context` tasks as we assume that condition testing does not impact resources.

- `[sequence:` $T_1,\ldots,T_m$`]`. T_i are tasks to be executed sequentially; the bound is $[\bigotimes L_i,\ \bigotimes U_i]$.

- `[parallel:` $T_1, \ldots, T_m$`]`. T_i are tasks to be executed in parallel; the bound is $[\sum L_i,\ \sum U_i]$.
- `[select:` $C_1T_1, \ldots, C_mT_m$`]`. The first T_i is chosen such that its condition C_i is satisfied. Since the C_i have no resource impact, the bound is $[\min L_i,\ \max U_i]$.
- `[wait:` $C\ T'$`]`. Task T' is executed once the condition C becomes true. Since we do not consider timing of tasks in this paper, and condition testing has no resource impact, the bound is simply that of T'.
- `[try:` $Q_1T_1, \ldots, Q_mT_m$`]`. The tasks Q_i are performed in order until some Q_s succeeds; the corresponding task T_s is then executed. Assuming that tasks fail at their end then a two-condition `try` statement expands to one of three sequences: $Q_1\ T_1$; $\mathbf{Q_1}\ Q_2\ T_2$; or $\mathbf{Q_1}\ \mathbf{Q_2}$, where bold denotes failure. The bound is the minimum and maximum of these possible sequences; for example, the lower bound is $\min(L_{Q_1} \otimes L_{T_1}, L_{Q_1} \otimes L_{Q_2} \otimes L_{T_2}, L_{Q_1} \otimes L_{Q_2})$.

Projection of replaceable resources through the iteration constructs `while` and `forall` is straightforward: if the projected resource estimate for a single execution of the loop body is $[L,\ U]$, then $[0,\ U]$ constitutes a guaranteed bound. Projection of non-replaceable resources for these constructs presents a challenge in that the number of iterations may depend on conditions or parameters that cannot be determined until runtime: for instance, looping over the number of people who reply to an invitation, or looping until a reply to an email request is generated.

The construct `[while:` $C\ T$`]` is particularly problematic, given the dependence on the dynamics of the termination condition C. Bounding the number of iterations requires both an upper bound d on the duration for which C is anticipated to hold, and a lower bound t on the time required to complete an iteration of T. If the projected resource estimate for a single execution of T is $[L,\ U]$, then $[0,\ U \times d/t]$ constitutes a guaranteed bound for the iteration.

Iteration based on the construct `[forall:` $x\ C(x)\ T(x)$`]` can be bounded in the following cases in which the exact number of iterations, n, can be determined. This occurs when (a) the quantification variable belongs to an enumerated type (e.g., iterate over people in the organization), (b) the iteration is defined over a concrete structure for which the number of elements can be ascertained (e.g., over a list), or (c) the scope of quantification is defined by a static predicate with bounded, known extent (e.g., purchase a flight for every member of the team). In these cases, if the projected resource estimate for a single execution of T is $[L,\ U]$, then $[L \times n,\ U \times n]$ constitutes a guaranteed bound for the iteration for a non-replaceable resource. By contrast, in the case when C is a dynamic predicate, precise determination of the number of iterations is not possible. If a bound v on the number of values for which C holds can be determined (e.g., register everyone who has a paper accepted), then $[0,\ U \times v]$ provides a guaranteed bound.

EXAMPLE 3. *Consider the task node* `arrangeCatering` *in Figure 2. Three possible plans could match the cue for this task: by brown bag, internal caterer, or external caterer. The plan for the last of these was shown in Figure 1. All three plans have resource approximations annotated as shown in Figure 2. For resource money, the approximations are functions of n, the total number of people to be catered, as before. For resource hours, the approximations are a single bound, and for licenses they are specified precisely. Since n is unknown prior to execution, the only computable approximation for money is the last, for example,* $[120,\ 1000]$ *for external catering. The resource aggregation produces the summary shown for node* `arrangeCatering` *from its three children:* $\langle$ (*money* $[20,\ 1000]$), (*hours* $[1,\ 3]$), (*licenses* 1) $\rangle$. *This information is propagated up the TE tree to its parent, the* `PARALLEL` *node. In the example, the number of visitors is a ground parameter to the root task: $v = 4$. Eventually, the resource summary derived for the root task node is* $\langle$ (*money* $[420,\ 1400]$), (*hours* $[9,\ 18]$), (*licenses* 3) $\rangle$.

By induction over the TE tree, the resource summary computed at the root node is a guaranteed enclosure of the actual resource consumption by the root task:

PROPOSITION 1. *Let T be a task and $[L,\ U]$ be the resource summary for resource R computed by the above algorithm on the TE tree rooted at T. Given that primitive tasks consume R within their resource approximations, then the amount of R consumed by T lies within $[L,\ U]$.* □

4. VARIABLE RELATIONSHIP GRAPH

Having described how to construct the TE tree and perform resource projection over it, we now describe the construction and use of the *variable relationship (VR) graph* for tracking relationships and constraints on variables. A VR graph enables both the propagation of parameter values that assist with computing resource bounds on tasks, and the pruning of impossible branches of a TE tree.

4.1 VR Graph Construction

We call the nodes of a VR graph *vertices*, to distinguish them from the nodes of a TE tree. Each vertex corresponds to a variable in a node of a TE tree. The (hyper-)edges of the VR graph correspond to relations between the variables.[1] Formally, the VR graph for a TE tree is a 4-tuple $\langle \mathcal{V}, \mathcal{D}, \mathcal{C}, \mathcal{E} \rangle$ where: $\mathcal{V}$ is a set of variables; $\mathcal{D}$ is a set of domains, one for each variable; $\mathcal{C}$ is a set of relations over $\mathcal{D}$; and $\mathcal{E}$ is a set of additional *possibility* relations, explained below. Thus $\mathcal{C}$ and $\mathcal{E}$ are two sets of constraints and $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \cup \mathcal{E} \rangle$ forms a classical *constraint satisfaction problem* (CSP) [1, 5].

The variables of a VR graph are divided into disjoint sets called *partitions*. Each partition corresponds to a different execution-time choice. These choices derive from two sources: (a) selection among alternative plan instances for a simple task (i.e., sibling plan edges in the TE tree), and (b) conditional branching via the `select` and `try` constructs. For each such choice, a partition is created to localize constraints and variables specific to that choice.

The two types of constraints, $\mathcal{C}$ and $\mathcal{E}$, represent relationships of different types between the variables. The edges in $\mathcal{C}$ encode constraints among variables *within* partitions; these constraints are derived from the predicates and functions used in the corresponding plans. In accord with terminology from the AI planning community, we say that two variables linked by an equality relation *codesignate* with each other. Second, the possibility relations in $\mathcal{E}$ link partitions. Each $\mathcal{E}$ constraint relates a variable in one partition to variables with which it may possibly codesignate in other partitions (depending upon which partition is chosen at execution time).

Consider a set of partitions created to represent selection among alternative plan edges for a simple task node T in the TE tree. For each task parameter `$x`, an edge in $\mathcal{E}$ represents the relationship from a vertex for `$x` to the vertices for the equivalent variables in the child tasks, as linked by the plan cue on the corresponding plan edge. For example, suppose T has two child nodes T_1 and T_2, with corresponding parameter variables `$x1` and `$x2`. The VR graph vertex for `$x` is related to each of `$x1` and `$x2` by a possibility relation. This describes that exactly one of `$x1` and `$x2` will codesignate with `$x` in the future, but which one is unknown at this time. This partitioning allows inference for different plan choices in the TE tree to proceed without cross-contamination from

[1] For simplicity, we assume (by introducing new variables, preconditions, and `context` tests) that the parameters in the cue of each plan and the parameters of the tasks in the body of a plan are all variables.

Algorithm 1 Pseudocode for VR graph constraint propagation (lines 1–10) and resource summary updating (lines 11–15)

```
 1: repeat
 2:    Choose a partition p such that V(p) is non-empty
 3:    Add the node N_TE(p) to Q
 4:    repeat
 5:       Pick a variable v ∈ V(p) and remove it from V(p)
 6:       for each constraint C involving v do
 7:          Update D(v') of each variable v' constrained by C
 8:          If any D(v') has changed, add v' to V(p_v)
 9:    until V(p) is empty
10: until V(p) is empty for all partitions
11: repeat
12:    Pick a node N ∈ Q and remove it from Q
13:    Update resource summary of N from the changed variable domains
14:    If resource summary of N changed, add the parent of N to Q
15: until Q is empty
```

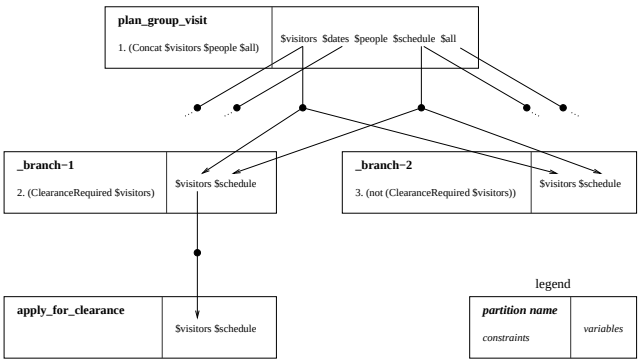

Figure 3: Variable relationship graph fragment

lower to upper levels; it also allows inferences from higher to lower levels, that is, from T to T_1 and T_2, for instance if the domain of `$x` is narrowed. If there is only one possible branch, then an equality rather than possibility relation links the variable vertices.

Similarly, each conditional branch (i.e., the `select` and `try` constructs) leads to a distinct partition with separate variables. For example, consider a plan containing the composite task `[select:` $(C_1$ `$x)` `[do:` $(T_1$ `$y)]` $(C_2$ `$x)` `[do:` $(T_2$ `$y)]]`. Separate partitions and new copies of the variables `$x` and `$y` are created for each branch. Possibility relations relate `$x` and `$y` to the corresponding branch variables $\texttt{\$x}_1$, $\texttt{\$y}_1$, $\texttt{\$x}_2$, and $\texttt{\$y}_2$. The partition for the first branch has the constraint $(C_1\ \texttt{\$x}_1)$; the partition for the second has the constraints $(C_2\ \texttt{\$x}_2)$ and $\neg(C_1\ \texttt{\$x}_1)$.

EXAMPLE 4. *Figure 3 shows part of the VR graph for the TE tree in Figure 2. The partition* `plan_group_visit` *contains the variables in the plan for the corresponding node in the TE tree, and a constraint derived from the* `context` *test in the plan. The* `select` *construct introduces two child partitions corresponding to the two branches, with constraints derived from the* `select` *conditions. Possibility relations link corresponding variables in parent and child partitions.*

The initial domain $D(v)$ of a variable v is derived from the type of v. We next describe how variable domains, and consequently resource summaries, are refined.

4.2 Propagation and TE Tree Pruning

Constraint propagation updates domains for variables in the VR graph, drawing both on information that is known when the TE tree is created, such as from variable domains and initial parameters, and information acquired during execution, such as when the domain of a variable changes or the graph is structurally modified. These updates in turn enable resource summaries to be refined as variables are instantiated or their domains are narrowed.

We perform a two-level constraint propagation that exploits the partitioned structure of the VR graph. The narrowing of a domain of a variable is propagated through the constraints of the partition containing the vertex for that variable, possibly resulting in the narrowing of other variable domains. Once a quiescent state is reached within a partition, the domain narrowings are propagated across the possibility relationships to neighboring partitions and the process is repeated. The domains are passed down to child partitions, but are merged with domains for the corresponding parameters in alternative partitions before being passed up.

Algorithm 1 gives the pseudocode for the VR graph constraint propagation and the subsequent updates to the resource summaries in the TE tree. In the pseudocode, let $V(p)$ be the set of variables in partition p that have changed and let p_v be the partition of variable v. Let $N_{\text{TE}}(p)$ be the TE tree node from which partition p is derived. Let Q be a queue of nodes in the TE tree, initially empty.

Lines 1–10 propagate domain changes through the VR graph constraints; lines 11–15 then propagate the effects in the TE tree. The loop starting at line 6 considers constraints in $\mathcal{C}$, which may affect the domains of related variables within partition p, and constraints in $\mathcal{E}$. Consider a hyperedge from variable v to variables $v_1, \ldots v_n$ representing a constraint in $\mathcal{E}$. If $D'(v)$ represents the new value of the domain of v after propagation, then $D'(v) = D(v) \cap \bigcup_i D(v_i)$ and $D'(v_i) = D(v_i) \cap D'(v)$.[2]

The narrowing of a variable domain to the empty set indicates that the containing partition is inconsistent. When propagated, this narrowing renders all descendant partitions inconsistent, and may make the parent inconsistent as well. A parent partition is made inconsistent if and only if all its child partitions are inconsistent.

Inconsistent partitions can be pruned from the VR graph; they can also be used to avoid expanding branches of the TE tree that are not viable. Moreover, if the constraints in one partition of the VR graph become inconsistent, given knowledge about possible variable bindings from the parent partition, the corresponding plan branch from the TE tree can be pruned.[3]

EXAMPLE 5. *Consider the VR graph in Figure 3. Since the top* `$visitors` *variable is bound to a specific list (in this example, of length four), the domain (not shown) associated with the variable is narrowed to a single value, that is, to that list. This narrowing is propagated to the domains of the* `$visitors` *variables in the branch partitions and then down to* `apply_for_clearance`*. The constraint associated with one of the branch partitions, say* `_branch_2`*, is now violated, leading to that branch being pruned. In that case, the unique value for* `$visitors` *allows refinement of the resource bounds on* `apply_for_clearance` *from* $[100, 500]$ *to the precise value* 300*, which is then propagated upward in the TE tree.*

Many agent and planning systems distinguish *static* predicates, which do not change over time, from *dynamic* predicates (i.e., fluents). Static predicates, such as equality and list membership, can be exploited to constrain variables in the VR graph, but because the truth value of dynamic predicates cannot be known until execution time, their usefulness for pruning the VR graph is limited. Hence, we do not reason about future values of dynamic predicates.[4]

[2] Note that for line 12 it is most efficient to handle all descendants of a node before handling the node itself. This means that Q should be partially ordered by $N_1 \prec N_2$ iff N_1 is a descendant of N_2 in the TE tree. Q can be kept sorted by inserting nodes in the correct place in lines 3 and 14.

[3] Note that this pruning may be more aggressive than in plan selection for an ordinary BDI agent: it may be possible using the VR graph to deduce that a plan must later fail due to constraint violations even if its preconditions are satisfied.

[4] Prior work in the planning community has investigated the use of the modal truth cri-

The partitioning of the CSP does not detract from the soundness or completeness of the inference, although it is worth noting that the agent's inference depends on the power of the constraint propagation techniques it has available, which we denote *Prop*. By standard results on *chaotic iteration* over the combined CSP [1]:

PROPOSITION 2. *Two-level constraint propagation over the VR graph $\langle \mathcal{V}, \mathcal{D}, \mathcal{C}, \mathcal{E} \rangle$, using Prop at each stage, achieves the identical outcome as Prop on the combined CSP $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \cup \mathcal{E} \rangle$.* □

5. RESPONDING TO EXECUTION EVENTS

A TE tree captures the anticipated resource usage for a task at a given instant in time. We have described how to construct TE trees prior to the start of task execution. In order to produce maximally informative bounds in the resource summaries, it is necessary to update the summaries at execution time as choices are made and events occur that could impact resource usage.

A typical BDI agent maintains its current intentions in data structures analogous to subtrees of the TE trees [13, 9]. As plans execute successfully, variables are bound to values, and these subtrees grow through the addition of task nodes. The VR graph tracks these changes by incorporating the variable bindings and pruning both the nodes in the TE trees and the partitions in the VR graphs that correspond to the choices *not* selected. In each of these cases, the additional information is propagated through the VR graphs and updated resource bounds are aggregated up the TE trees.

EXAMPLE 6. *Consider the point at which the* `arrangeCatering` *task of Figure 2 is to be executed. When the plan corresponding to the* `arrange_catering_external` *node is selected, the sibling nodes can be pruned, leading to a more precise resource bounds estimate. For the resource money, for instance, supposing the value of n is not yet known, the bound at the node* `arrange_catering` *becomes* $[120, 1000]$.

The effects of successful execution are *monotonic* in that they lead only to refinement, not expansion, of previously computed resource bounds. This monotonicity results because (a) the operations do not invalidate previous variable bindings or constraints, and prune only those parts of the VR graph that are certain not to impact resource consumption, and (b) elimination of options can only narrow, not expand, bounds. In contrast, task failures can lead to nonmonotonic changes to resource bounds, since an agent must reconsider its plans and may potentially prune failed subtrees in the TE tree in favor of previously rejected alternatives. Such failures require the recomputation of the resource bounds for the impacted portions of the TE tree.[5]

6. EXPLOITING SEMANTIC KNOWLEDGE

This section presents two further types of execution-time refinements to resource projections. Both leverage additional semantic knowledge about relationships among task parameters.

6.1 Active Solicitation of Requirements

In certain situations, an agent may actively gather information to provide explicit estimates for the resource requirements of a task. For example, the cost of a reception would generally be solicited as part of the process of organizing a workshop. We call a task that produces such an upper estimate a *resource estimation task*.

When a resource estimation task produces an estimate e for a given task T, the prior estimate $[l, u]$ for T can be refined to $[l, e]$. In some cases, it may even be possible to obtain precise values, as is the case when booking a hotel. We call a task that produces a precise characterization of resource consumption a *resource bid task*. When a resource bid task produces a bid b for T, the bounds for T can be set to $[b, b]$.

In certain cases, it may be appropriate for the agent to obtain multiple estimates for a resource-consuming task. If all such estimates $E = \{e_1, \ldots, e_m\}$ can be obtained together, then the resource bounds for the task can be set to $[\min E, \max E]$.[6]

6.2 Selection and Narrowing

The relationships among variables tracked in the VR graphs derive from codesignation constraints among variables both within and across plans (the latter arise from plan application to a task). Further inference can be enabled by augmenting the VR graphs to exploit additional types of inter-variable relationships. Here, we describe two common task idioms that produce such relationships, and indicate how a VR graph can be extended to reflect those relationships and support additional execution-time refinements of resource bounds.

The first idiom focuses on *selection* tasks. Consider a task (Select `$c` `$s`), where `$c` represents a set of values under consideration for some role, and where execution of the task results in the binding of the variable `$s` to a value for that role. The binding of `$c` to a list of values restricts the domain of values for `$s` to the same list. Although the binding for `$s` is not yet known, reasoning by cases can be applied over the possibilities, *before* the task (Select `$c` `$s`) is performed, to any resource approximation for a downstream resource-consuming task that depends on a parameter linked from `$s`. However, such a restriction on the domain of `$s` cannot be exploited within the VR graph as formulated, since no explicit constraint connects the variables `$c` and `$s`.

The second idiom focuses on tasks that perform some kind of *narrowing* operation that reduces the domain of a variable that is used subsequently to provide a resource approximation for a task. That is, the task reduces some *source* domain to a smaller *result* domain. For instance, in the flight purchase example given in Sect. 1, one narrowing operation would be the decision to consider only budget airlines, which precludes tickets from full-service carriers and thus refines the set of candidate flights.

To enable these forms of projective reasoning, we extend a VR graph as follows. We require semantic annotations that identify *selection* tasks along with their *candidate* and *selection* parameters, and *narrowing* tasks with their *source* and *result* parameters. Let (Member `$e` `$s`) define the constraint `$e` $\in$ `$s` and (Subset `$s1` `$s2`) define the constraint `$s1` $\subseteq$ `$s2`. We extend the constraints $\mathcal{C}$ of the VR graph as follows:

- If $T(x_1, \ldots, x_n)$ is a selection task with candidate parameter x_c and selection parameter x_s, then add (Member x_c x_s)
- If $T(x_1, \ldots, x_n)$ is a narrowing task with source parameter x_s and result parameter x_r, then add (Subset x_r x_s)

Exploitation of these additional VR graph links involves a straightforward generalization of the propagation algorithm of Sect. 4.2; in the interest of space, we forego a detailed description here.

teria for predicting the future state of dynamic predicates that model conditions under control of the system [2]. Many dynamic predicates, however, represent exogenous domain properties that lie beyond the agent's control (e.g., temperature, time of arrival of a flight) and so cannot be predicted using those methods.

[5] Thangarajah et al. [16] point out that, depending on the intended use of resource information, it may be possible to update data structures such as TE trees lazily, i.e. mark them as 'dirty' when change occurs, but not compute the new values until required.

[6] When estimates are solicited independently, there is an operational issue of knowing when the last estimate has been obtained so that the revised bound can be computed. In particular, if bounds are computed prematurely they may be overly narrow, thus violating our requirement for guaranteed bounds. Static analysis to determine the last estimation task is made difficult by the temporal unpredictability of asynchronous threads and loops, and the possibility of conditional branching.

7. DISCUSSION AND RELATED WORK

Our task expansion tree and associated aggregation of resource summaries are similar in spirit to the *goal-plan tree* of [16]. The latter is composed of alternating layers of goal nodes (analogous to our simple task nodes) and *plan nodes*; their plans are limited to sequential or parallel execution of propositional tasks. The same authors reason with a failure model whereby, if a plan T_1 fails for a task T, then every other possible plan $T_2, \ldots, T_m$ to achieve T is attempted in turn, until one succeeds or all have failed (when T is said to have failed). This failure model is reflected in their construction of the goal-plan tree, but is overly conservative and so may limit task throughput (and, hence, agent effectiveness). Last, the focus of [16] and their related work is to determine when tasks might be in potential conflict, and to resolve conflicts where possible by scheduling the task executions.

We have described a limited form of lookahead, with respect to resources. de Silva and Padgham [4] go further in deliberately incorporating full AI planning over possible plans (recipes) within an agent architecture, by the action of invoking a hierarchical planning system. Techniques for incorporating resource feasibility testing into hierarchical planning have been investigated previously [6, 2]. These approaches perform a kind of static analysis similar to that performed by our aggregation over the task expansion tree, although they support replaceable resources and resource production through the incorporation of a modal truth criteria analysis. The work assumes a single known value or single bound for the resource requirements of individual tasks, in contrast to our multi-fidelity approach. Furthermore, they do not address the issue of execution-time refinement of aggregate resource bounds.

Resource estimation is well studied in scheduling and temporal reasoning. For instance, Laborie [8] describes propagation of resource constraints in constraint-based scheduling, while Muscettola [10] computes an *envelope* encompassing the predicted evolution of resource levels over time. The emphasis in scheduling is to compute ahead of execution a feasible schedule that guarantees no resource conflicts. In contrast, detecting plan failure and recovering from it is an integral part of a reactive agent. Thus our projection of resources need not be complete; it has value even when partial since any sound inference is informative to the agent. Nonetheless, techniques particularly from constraint-based scheduling hold potential for extending our work to consider predicting resource evolution over time. Such work also points to the scheduling of tasks to avoid possible resource conflicts, as also considered by [16].

8. CONCLUSION

The framework for resource projection presented here provides a pragmatic middle ground, combining the static computation and dynamic refinement of guaranteed bounds on projected resource usage by a BDI agent. Our framework is distinguished by its use of parameterized multi-fidelity models of expected resource consumption, its focus on maximally informative resource bounds, and an expressive plan language that includes parameterized task types.

We have implemented our resource projection framework in the SPARK system, and are applying it to real-world domains derived from the CALO cognitive assistant project [14]. These domains include a full version of the group visit example in this paper, domains describing purchase of office equipment, and registration and travel to a conference. Construction of the TE trees and associate VR graphs, constraint propagation on the variables, and initial pruning of the trees are all performed with a few hundred milliseconds of runtime; the overhead is in line with required performance of CALO as a mixed-initiative task execution assistant. In addition to the minimal overhead of the static resource projection, the dynamic updates employ CSP propagation techniques of low-order polynomial complexity [5], and thus scale effectively.

Our work enables improved task throughput and execution quality, with reduced failure. It is straightforward to construct scenarios in which our approach will substantially outperform existing techniques. However, a truly meaningful evaluation of benefits will require analysis of extended agent operations within a real (or realistic simulated) environment. We will be collecting such data as our system is deployed within the CALO user community.

The work presented here focuses on guaranteed bounds for resource consumption. An interesting next step would be to develop a probabilistic counterpart to this framework that establishes the *likelihood* of resource contention by incorporating stochastic models of event likelihood and task success, building on a framework such as the probabilistic representation for process models of [12].

As noted earlier, future work is to consider a broader range of task resource consequences, including knowledge post facto costs and nondeterministic resource consumption. In addition, while much of the work described in this paper applies to production as well as consumption (on which we have focused), resource production and its interaction with consumption requires a model with temporal extent, by which the agent can reason on the relative timing and interleaving of production and consumption, including the return of replaceable resources.

Acknowledgments. We thank the reviewers for their comments. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA), through the Department of the Interior, NBC, Acquisition Services Division, under Contract No. NBCHD030010.

9. REFERENCES

[1] K. R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1–2):179–210, 1999.

[2] B. J. Clement, A. C. Barrett, G. R. Rabideau, and E. H. Durfee. Using abstraction in planning and scheduling. In *Proc. of ECP'01*, 2001.

[3] M. Dastani and L. van der Torre. Specifying the merging of desires into goals in the context of beliefs. In *Proc. of EurAsia ICT '02*, 2002.

[4] L. P. de Silva and L. Padgham. Planning on demand in BDI systems. In *Proc. of ICAPS'05 Poster Session*, pages 37–40, June 2005.

[5] R. Dechter. *Constraint Processing*. Morgan Kaufmann, San Francisco, CA, May 2003.

[6] B. Drabble and A. Tate. The use of optimistic and pessimistic resource profi les to inform search in an activity based planner. In *Proc. of AIPS'94*, pages 243–248, Chicago, IL, June 1994.

[7] M. P. Georgeff and A. L. Lansky. Procedural knowledge. *Proc. of the IEEE*, 74(10):1383–1398, 1986.

[8] P. Laborie. Algorithms for propagating resource constraints in AI planning and scheduling. *Artificial Intelligence*, 143(2), 2003.

[9] D. Morley and K. Myers. The SPARK agent framework. In *Proc. of AAMAS'04*, pages 714–721, New York, NY, July 2004.

[10] N. Muscettola. Computing the envelope for stepwise-constant resource allocations. In *Proc. of CP'02*, pages 139–154, Sept. 2002.

[11] K. L. Myers and N. Yorke-Smith. A cognitive framework for delegation to an assistive user agent. In *AAAI 2005 Fall Symposium on Mixed-Initiative Problem-Solving Assistants*, Nov. 2005.

[12] A. Pfeffer. Functional specifi cation of probabilistic process models. In *Proc. of AAAI'05*, pages 663–669, Pittsburgh, PA, July 2005.

[13] A. S. Rao and M. P. Georgeff. Modeling agents within a BDI-architecture. In *Proc. of KR'91*, pages 473–484, 1991.

[14] SRI International. CALO: Cognitive Assistant that Learns and Organizes. `www.ai.sri.com/project/CALO`, Mar. 2005.

[15] J. Thangarajah, L. Padgham, and J. Harland. Representing and reasoning for goals in BDI agents. In *Proc. of the Australasian Conference on Computer Science*, Melbourne, Australia, Jan. 2002.

[16] J. Thangarajah, M. Winikoff, L. Padgham, and K. Fischer. Avoiding resource conflicts in intelligent agents. In *Proc. of ECAI-02*, 2002.