Agent Independent Task Planning†

William S. Davis
Boeing AI Center
PO Box 240002, MS JA-74
Huntsville, AL 35824-6402
net: bill@huntsai.boeing.com

ABSTRACT

Agent-Independent Planning is a technique that allows the construction of activity plans without regard to the "agent" that will perform them. Once generated, a plan is then validated and translated into instructions for a particular agent, whether a robot, crewmember, or software-based control system. Because Space Station Freedom (SSF) is planned for orbital operations for approximately thirty years, it will almost certainly experience numerous enhancements and upgrades, including upgrades in robotic manipulators. Agent-Independent Planning provides the capability to construct plans for SSF operations, independent of specific robotic systems, by combining techniques of object-oriented modeling, nonlinear planning and temporal logic. Since a plan is validated using the physical and functional models of a particular agent, new robotic systems can be developed and integrated with existing operations in a robust manner. This technique also provides the capability to generate plans for crewmembers with varying skill levels, and later apply these same plans to more sophisticated robotic manipulators made available by evolutions in technology.

1. Introduction

Space Station Freedom is planned for orbital operations for approximately thirty years. Over the long life of this complex structure, it will almost certainly experience numerous enhancements and upgrades, including upgrades in robotic manipulators. Great potential for robotic automation exists in the areas of housekeeping, laboratory science, maintenance, and safety, as well as various EVA functions. Throughout this 30-year period the types of robotic manipulators available for these areas, as well as the capabilities they provide, will continuously evolve with changes in technology. On the contrary, basic procedures for intra- and extra-vehicular activity, once established, will remain relatively static. As advances in technology produce more sophisticated manipulators that are capable of performing more complicated tasks, robots may become responsible for more detailed operations. However, for these advancements in technology to be beneficial to Space Station Freedom, any robotic upgrades should be *compatible* with existing procedures.

Programming different robots for the same task is a redundant job that should be avoided. Such programming can be very labor-intensive, not to mention the job of verifying that the "new" robots are still compatible with the "old" tasks. Some tedious chores that crewmembers perform today are the duties that may be carried out by the robots of tomorrow. Thus, the robotic plans that will be developed for future on-board robots should be compatible with crew procedures that are established in the interim. Persons who are currently responsible for composing such crew

procedures may eventually be tasked to compose these procedures for robots as well. In this regard, prescribing activities for robots should be as similar as possible to prescribing activities for crewmembers.

A technique known as "Agent-Independent Planning" has been developed for addressing the above issues. Planning is determining a course of action to achieve some goal. Task planning is determining a sequence of tasks (the course of action) to be performed by an agent to achieve some desirable state (the goal) in the agent's world. "Agent-Independent Planning" is a method of automated planning that allows the generation of task plans from a set of goals, without having to be concerned with constraints imposed by the agent that will execute the plan. In the domain of Space Station Freedom (SSF), these plans can be considered a sequence of tasks for intra-vehicular and extra-vehicular operations activity. Plans, or operations procedures, are developed by considering general constraints on the planning environment and task sequences. For execution of these procedures, the plans are translated into the specific operations language of a particular agent. This methodology allows plans and their environment to be modeled in a fashion that separates different classes of constraints into independent sets.

A prototype of such a system has been developed at Boeing Aerospace and Electronics that creates agent-independent plans for SSF maintenance and repair operations. The system translates these plans into (a) code which is executed by a robot, (b) software commands which drive a graphical robotic simulator, or (c) English sentences (output through a voice synthesizer) which describe crew procedures. The actual planning mechanism is based on Chapman's TWEAK [3], but the representation incorporates Allen's time logic [1] and hierarchical abstraction [8]. Hogge integrates a temporal interval-based planner in the domain of Qualitative Physics [4], compiling plan operators by matching descriptions of an agent with descriptions of the domain. The system presented in this paper performs along similar lines to match an agent's capabilities with the needs of a plan. This combination is validated to ensure the agent can perform the given plan.

This paper begins with an explanation of the plan representation in terms of the object-oriented model of the plan environment, the temporal relationships among tasks in the plan, and how these two representations are abstracted into a task library. Once this foundation is established, a discussion ensues concerning the generation of agent-independent plans, and the steps necessary to translate these plans for a specific agent.

2. Plan Representation

In order to have agent-independent planning, one must develop a representation for the plans which is free from the agent who will perform the plan, but also which can later be transformed into a representation that includes the agent. Such a representation must model the world in segments which can be connected and disconnected to identify various aspects of the world. We accomplish this by decomposing the planning environment into three distinct entities: Tasks, Agents, and Objects. Thus, tasks are actions on objects, agents are the performers of actions, and objects are the recipients of actions. The agent-independent plan is built combining tasks and objects, and then the transformation is made to agent-dependency by incorporating knowledge about a specific agent. Obviously, "action" is the common denominator of these three entities. In a robotic realm, these actions represent physical motion. A primitive action is defined to be that which represents basic physical motion, such as locomotion, rotation, limb movement (extension, retraction, lateral/horizontal/diagonal movement), etc. Figure 1 lists the primitive actions currently employed in the planning environment. Models of the planning environment (whether agents, objects, or tasks) all relate to this set of primitives. These actions, or primitive tasks, provide the common

interface between agents and the planning environment. The agents' and objects' physical and functional constraints are represented in terms of preconditions and effects on these primitive tasks. Object-oriented models of agent and object properties allow descriptions of agents and objects to be combined through inheritance, as will be shown in the following section.

rotate clockwise	move to	move left	grasp	push	raise
rotate counter-clockwise	carry to	move right	release	pull	lower

Figure 1. Primitive Actions Employed in Planner

2.1. Object-Oriented Environment

Agents and objects are composed of properties which are ordered hierarchically, with lower levels inheriting the properties of higher levels, and communication between them is done in message-passing fashion. This hierarchical ordering allows complex, real-world descriptions of items in the environment. In terms of the Space Station Freedom maintenance domain, this means that complex items such as thermal control systems can be described by the union of the properties of their components (such as pumps, valves, pipes, etc.). Tasks, agents and objects are modeled with primitive actions as a "connecting point". In terms of this paper, task information declares which objects are affected by action (and possibly in what order the action is to occur), information about agents declares which actions they can perform and to what capacity, and object information specifies the manner in which objects can be affected by actions.

Object: pcb / Task: PUSH(Distance?)	Object: pcb / Task: PULL(Distance?)
subgoals:	*subgoals*:
pcb . attached-p = no	pcb . attached-p = yes
Slot? . occupied = no	Slot? . parent-object = P-Obj?
Slot? . parent-object = P-Obj?	P-Obj? . power-status = off
P-Obj? . power-status = off	pcb .attached-to = Slot?
pcb . coordinates = Slot? . observation-pt	pcb . depth = Distance?
pcb . depth = Distance?	*main-effects*:
main-effects:	pcb . attached-p = no
pcb . attached-to = Slot?	*side-effects*:
side-effects:	pcb . coordinates = Slot? . observation-pt
pcb . attached-p = yes	pcb . attached-to = nil
pcb .coordinates = Slot? . coordinates	Slot? . occupied = no

Figure 2. Functional Constraints for Printed Circuit Board

Agents and objects are represented as a mixture of characteristics describing their physical properties and functional capabilities. Such physical properties include size, mass, relative position, etc. while functional capabilities are constraints based on the primitive actions, as well as physical properties of agents and objects. These functional constraints are based on typical planning constraints, such as the preconditions/subgoals and effects found in Wilkins' SIPE [11]. This allows each agent or object to respond differently to the same primitive action according to the manner in which it is modeled. Figure 2 shows an example of the object constraints for pushing or pulling a printed circuit board. It is interesting to note how the constraints work to somewhat

specialize these generic actions into "remove" and "insert" actions. Pattern-matching is employed to allow generic constraints to be tailored for specific objects. In the figure, variables are represented by symbols appended with a question mark. These variables are instantiated in the planning process, and then the constraints are used to determine truth satisfaction for including the action in the plan. Functional constraints on an agent work in a similar fashion.

Also included in the definition of an agent are the specific instructions necessary for it to perform a primitive action. It is this set of instructions that will be used to transform the plan representation from agent-independent to agent-dependent. Agent characteristics are arranged in a hierarchy such that properties of higher level are inherited by lower levels. For example, in figure 3 all the properties associated with limbs of motion are inherited by both arms and legs. In turn, arms and legs add their own distinguishing properties and capabilities which are inherited by their lower levels (which inherit the things from the limbs of motion characteristic as well). Thus to describe a Puma-560 robot with one arm, incorporating vision, force/torque, and tactile sensors, one needs to include in the robot's description the corresponding characteristics and then add the features particular to the Puma-560 basic unit. Objects are represented in similar fashion. However, whereas agents were classified on the grounds of their capabilities to effect action, objects are classified according to how they are affected *by* action; the difference is one of *activity* versus *passivity*. For instance, the object classifications, or characteristics, for a printed-circuit-board (which is mounted on some rack) would include "replaceable object" and "fragile object". "Replaceable object" would be derived from "removable object" (which has constraints that inhibit certain actions on the object when "in-place"), while adding its own information concerning the location of replacement objects, the location to discard used objects, etc.

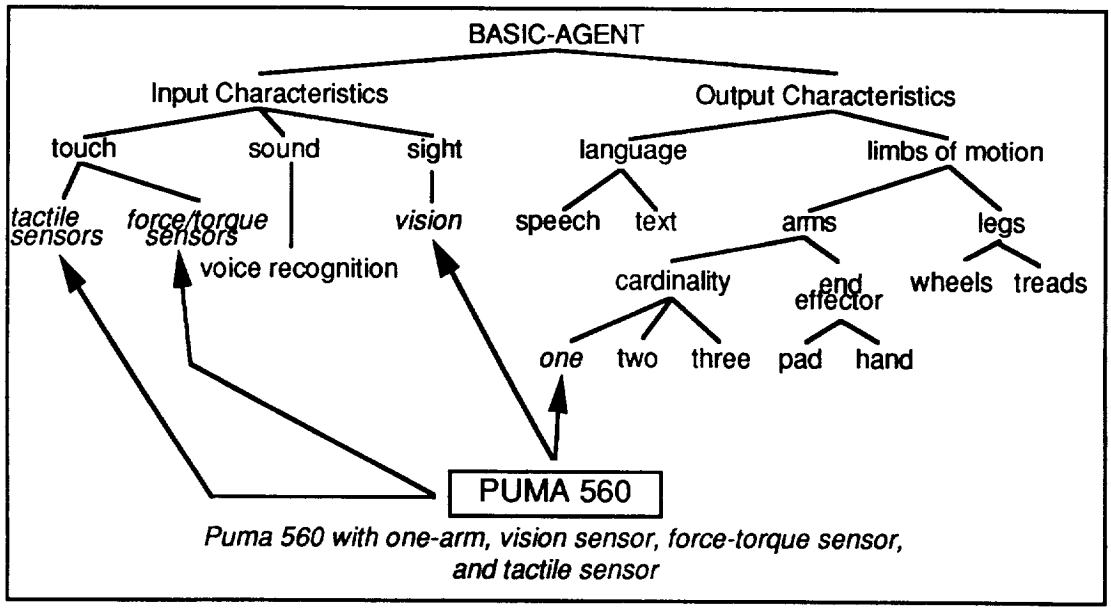


Figure 3. Example Agent Hierarchy

2.2. Task Abstraction

As was stated earlier, tasks declare which objects are affected by action. Actions are composed of sub-actions. For instance, *replace pump* may be broken into *isolate pump*, *remove pump*, *discard pump*, *get a new pump*, and *install pump*. Primitive actions (see figure 1) correspond to primitive tasks, and are used to create higher-level tasks. Higher-level tasks are

composed of previously defined higher-level and/or primitive tasks arranged in some fashion (see section 2.3 for this arrangement). Thus, a (theoretically) infinitely large task library can be developed using this means of abstraction. This capability provides for a dynamic planning environment. That is, the plan that solves some goal today may well be composed of different tasks when trying to solve the same goal tomorrow (assuming new tasks are being added to the library). This allows the system to improve incrementally. By introducing plan variables, tasks then become predications on objects as they are parameterized, such as *remove(pump-27)*. The argument to a task can also be an object characteristic (as defined above) in order to describe generic, or template, tasks for use in describing more abstract tasks. This is analogous to the manner in which Wilkins [11] uses constraints to construct partial descriptions of objects, as well as the specialization of abstraction in Tenenberg's [8] plan graphs.

When instantiating plan variables to actual objects or object characteristics, the constraints associated with that object are added to any existing constraints with the task. When abstracting a group of tasks into a higher-level task, constraints are combined into a single set representing the abstracted action. Plan variables which share the same symbol are made to be codesignating, and any preconditions which are not satisfied within the abstracted group remain as a precondition for the higher-level task; similarly, effects which reach outside the scope of the task group are established as an overall effect for the higher-level task.

As initially stated, a plan is a set of tasks to be performed in some order. Because a higher-level task fits this description, a plan is merely a higher-level task (throughout the remainder of this paper, "task" and "higher-level task" will be used interchangeably). Each plan's sub-tasks are ordered according to their connections, which represent sets of operators from temporal logic specifying their sequence within the plan.

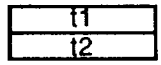
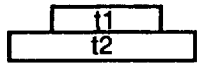
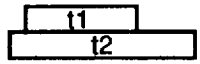
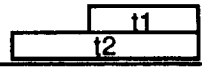
Relation			Operator	Inverse Operator	Pictoral Example
t1	*before*	t2	<	>	
t1	*equal*	t2	=	=	
t1	*meets*	t2	m	mi	
t1	*overlaps*	t2	o	oi	
t1	*during*	t2	d	di	
t1	*starts*	t2	s	si	
t1	*finishes*	t2	f	fi	

Figure 4. Possible Temporal Relationships

2.3. Temporal Plan Network

The temporal operators that reflect time relationships between tasks are those discussed in [1], and are summarized in figure 4. The temporal relationship between two tasks is expressed as a disjoint set of temporal operators. Hence, *(task1 [<, s, o] task2)* denotes that task1 either *is before, starts simultaneous to*, or *overlaps* task2. Using such sets as links between tasks, we construct a temporal network whose nodes are tasks and whose connections are temporal relationships between them. The network is constructed regardless of the ability of the agent who will ultimately perform

the tasks. We need not worry about parallelism during task/plan description, nor about whether the agent has one, two, or even ten arms. The only concern is what relationship each task has to other tasks. Undefined relationships are inferred based on defined relationships. Figure 5 shows an example; if we have the relationships *(task1 [s] task2)* and *(task2 [o] task3)* defined, we infer the relationship *(task1 [<,m,o] task3)*.

With the addition of temporal information, task abstraction becomes similar to the concept of reference intervals in [1]. Each higher-level task, then, is a temporal network of tasks from lower levels. For any such task, the temporal relations among its subtasks are validated by maintaining their transitive closure, which prevents ill-definitions such as *(task1 < task2)*, *(task2 < task3)* and *(task3 < task1)*. This also may reduce some of the explicit ambiguity expressed in the task's definition. Any remaining ambiguity is resolved during plan translation (as seen in the next section). By means of task abstraction, expressed ambiguity, and plan variables (which can either be instantiated to actual objects, or be constrained by object characteristics), non-trivial plans can be constructed that are completely independent of the agent that will perform them.

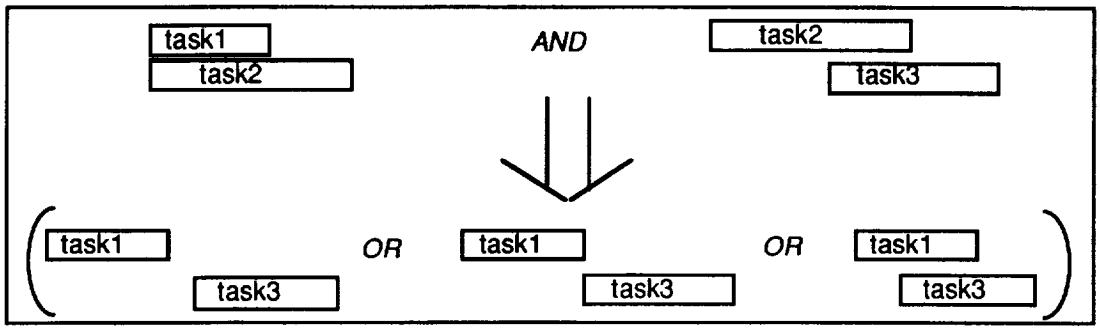


Figure 5. Implication in the Temporal Logic

3. Plan Translation

3.1. Validation Under Agent Constraints

In order to execute some plan, a declaration must be made as to what agent should perform the plan. Note, however, that this declaration may specify an agent that is incapable of such action. Therefore, the combination of agent and plan must be *validated*. This is essentially the job of plan translation. The luxuries that afford representation of the plan free from knowledge about an agent must now be considered in light of the agent capabilities. That is, any ambiguous part of the specified plan (plan variables, task abstraction, temporal ambiguity) must be resolved to a point where specific instructions understandable by the agent can be generated. While the agent-independent plan is represented in a hierarchy of temporal networks containing all possible orderings of plan performance, plan validation attempts to eliminate those temporal possibilities which are infeasible for a given agent, effectively producing an agent-dependent network. The validation process ensures that at least one possible traversal through this agent-dependent network is an acceptable plan, according to the agent's constraints, for accomplishing the desired goal. It is at this point that primitive tasks are mapped to specific agent instructions for execution. If at any point the agent is found incapable of performing the plan, it is said to be "rejected" from translation.

The first step of translation is to assign to the plan variables specific objects which the plan will manipulate, making use of object characteristics (see section 2) to aid in constraining the assignment. Once all the actual objects are determined, they can be used to check the physical

constraints of the specified agent. This is done by "matching" the physical properties of the agent with those of the objects. For instance, one constraint checks the mass of an object against the possible mass movable by the agent, while another constraint checks the size of each object to be sure the agent possesses the proper tools and end effectors to manipulate the object. Such constraints are used in validating that the agent is physically capable of manipulating the objects.

Once the agent's physical capability to perform the plan is established, the next matter of validation is to ensure the agent is functionally capable of plan execution. Functional constraints of an agent specify the temporal capability of an agent. Once a particular agent is designated for translation, its functional constraints (in the form of planning preconditions and effects) are integrated with the existing task and object constraints in the agent-independent plan. An agent is found functionally incapable when the plan requires simultaneous action which the agent cannot perform. A second reason for rejection is simply an ordering of the tasks that is incompatible with the agent's ability at a given moment. Examples of this are directing the agent to "grasp the pump" when no "hand" is free (i.e., is empty and available), or telling the agent to "release the filter" when no filter is being held. The temporal check made by functional constraints must examine transitions from one primitive task to another, and the timely ordering of the tasks can certainly be confirmed by stepping through the plan while "simulating" the action in the world.

3.2. Deriving an Agent-Dependent Network

An agent-dependent network is produced essentially by eliminating any infeasible temporal possibilities in the plan ordering. Presented here are two methods for obtaining this network. The first results in a network which contains every possible ordering in which the agent can successfully perform the primitive tasks to satisfy the goal, and excludes any problematic orderings. While this method has exponential computational complexity, it performs satisfactorily for small networks. Plans of significant abstraction, however, demand a more efficient translation process. The second method uses a simple heuristic that reduces the algorithmic complexity; and although the resulting network may produce an incomplete list of viable orderings, a straightforward assumption reducing the impact of this deficiency makes this method a preferred alternative to the first.

The essence of the first method to derive an agent-dependent network lies in computing the transitive closure over the temporal operators in the agent-independent network. Recall that such a transitive closure is always maintained for the subtasks of any high-level task while it is being defined. With this in mind, computing the transitive closure of *all* primitive tasks is a matter of relating tasks across hierarchical boundaries. This can be done using the "during" relationships (starts, during, finishes, and corresponding inverses) to relate sibling subtasks, as is done in the reference hierarchies of [1]. Such transitivity applies through all levels of the plan hierarchy, and can thus relate any primitive task to any other primitive task within a plan. A backtracking algorithm as is mentioned in [1] can be used to minimize ambiguity among the temporal operators in the final "closed" network. Once established, an agent-dependent network can be derived by a breadth-first-like traversal of the independent-network, updating virtual copies of the objects to maintain the status of the world, and eliminating operators that cause the traversal to "back up" due to an untimely ordering of events in consideration of the agent.

Intuitively, the above procedure operates in exponential time; a transitive closure algorithm for temporal intervals is discussed in [10]. The observation can be made, however, that a vast majority of these inferred links are irrelevant to the overall order of the plan. This raises the question: which links are necessary in validation and which are not? The temporal links of importance are those which impose simultaneous action on the agent. This means that validation

should be concerned not so much with the interconnections between *all* of the tasks, but rather with those which reflect the transition from one task to another. This focus on transitional tasks at the primitive level is the heart of the second method for obtaining an agent-dependent network.

This "endpoint method" makes use of a couple of existing properties of the agent-independent network to save time. First, since temporal links among the individual task's subtasks are consistent due to the maintenance of the transitive closure throughout task description, the entire network is temporally consistent before validation/translation begins (i.e., consistency among parents and consistency among children => consistency among siblings). Therefore, the transitive closure is unnecessary for this purpose. Second, by the above assumption, transitory activity is of main importance, so this method only considers relationships between primitive tasks which possibly occur at endpoints of higher-level tasks. Starting with the level-1 tasks (one level above the primitive/leaf level), a *begin-set* and *end-set* is computed for each set of subtasks. These represent the tasks that can possibly start or finish their parent task, respectively. Given higher-level tasks T1 and T2, and their respective begin-sets and end-sets, the "during" relationships are used to infer relationships among members within these sets. However, unlike the previous method which established relationships among all the primitive tasks, this method only uses the endpoint relationships to modify higher-level relationships pre-existing in the agent-independent plan. That is, the temporal relationship between two tasks T1 and T2 is pruned to eliminate any conflicts in transitioning from a primitive subtask of T1 to a primitive subtask of T2. Once the endpoint relationships are considered at this level, begin and end sets are computed for the next higher level in similar fashion, considering further relationships at the primitive level, and pruning those relationships from the network that are invalid for the agent. Pruning occurs either at the "local" network (the endpoint level), eliminating the temporal operator(s) which caused a conflicting task to be considered as an endpoint, or at the most abstract level that is appropriate, eliminating the temporal operator(s) that caused the endpoint sets to be in conflict. Finally, this method differs from the previous one, which used breadth-first traversal to find *all* viable paths, by using depth-first traversal to find *a* viable path. The viable path found becomes the sequence of the agent-dependent plan.

4. Plan Generation

Although the primary intent of this paper is to present "agent-independency," a brief description of the actual planner is provided to show how it uses the various planning and temporal constraints. The planner is nonlinear; that is, it produces a plan by deriving and further constraining sets of partial orders. It incorporates a constraint posting theme, using the objects' and agents' functional constraints as planning constraints. Certainly more thorough discussions of planning constraints and techniques exist elsewhere [3,11], but the terms are briefly defined here for clarity.

Each task contains an associated set of planning constraints (preconditions, subgoals, main effects, and side effects), which result from combining any associated agent or object constraints with other abstracted constraints. *Subgoals* are those conditions which must be satisfied before the execution of a task. *Preconditions* are those conditions which must be satisfied before the inclusion of that task into the plan. In essence, they are subgoals that are immediately satisfiable from tasks already existing in the plan. *Main effects* are conditions resulting from a task, and serve as a reason for selecting a task to achieve a particular goal. *Side effects* are conditions which are caused by a task, but which are not significant enough to warrant its inclusion in the plan. When adding tasks to a plan, a *clobberer* is a task which potentially defeats a precondition for another task, thus causing a break in plan causality. *Promotion* is the technique of constraining the clobberer to occur after the time when the precondition must be met. *Separation* is the technique of constraining the

clobberer not to codesignate with the precondition. That is, any plan variables that could be instantiated such that the clobberer's effects would defeat the precondition are prevented from doing so. Using these terms, figure 6 presents an outline of the planning approach.

Using the simple concepts of promotion and separation as defined above will force the planner to build plans that are sequential at their most abstract level. Incorporating temporal properties into the planner allows the construction of plans that are not committed to any particular order. Work along the lines of [2,9] associates temporal intervals with each task and condition/effect. Goals are achieved by "collapsing" intervals (asserting them be "equal") of conditions and effects. Hence, when a task is inserted into the plan its main effect is asserted to be temporally equal with the subgoal it is supposed to satisfy. This new relationship is propagated throughout the (top level) tasks in the plan. Upon detecting a clobberer, promotion and separation can still be employed, but rather than imposing a sequential constraint the conflicting tasks are simply constrained not to share the same interval (i.e., not be overlapping, during, starting, etc.). Such a combination between traditional nonlinear planning and temporal planning allows the generation of task plans that make no presupposition concerning which activities can be parallel and which must be sequential.

General procedure to satisfy a goal:

1. If a main or side effect which satisfies the goal already exists in the plan, then constrain the effect's task to precede the goal. (and procede to next goal)
2. Otherwise, select a task whose main effect matches the goal and whose preconditions can be immediately satisfied. (fail if no such task exists)
3. Insert task *T* into the plan, binding any plan variables necessary, by constraining *T* to precede the goal and constraining tasks satisfying *T*'s preconditions to precede *T*.
4. If any clobberers exist, try promoting them past the clobberee. (otherwise step 7)
5. If promotion fails, try separation.
6. If separation fails, then backtrack to step 2 and select a different task to satisfy the goal.
7. Upon successful addition of *T* into plan, place *T*'s subgoals onto goal queue and continue until goal queue is empty.

Figure 6. Outline of Planning Approach

5. Summary and Future Directions

A system has been introduced for describing and generating plans in a representation independent of an agent, which can subsequently be translated into agent-dependent instructions suitable for execution. Agents and objects are represented in an object-oriented fashion, allowing their description of physical and functional capabilities to assist in constraining plan description/generation, and to be "matched" for plan validation purposes in translation. Tasks can either be primitive actions based on movement, or abstracted to higher-level tasks (synonymous with plans) consisting of subtasks arranged in "temporal networks" (which allow temporal ambiguity in describing task orderings). Tasks and objects are combined to form an agent-independent plan. Plan translation transforms this into an agent-dependent plan by validating the combination of the plan with the properties and constraints of a specific agent, resulting in a set of instructions executable by the agent. Plans are constructed automatically using nonlinear planning techniques which operate on functional constraints of agents and objects. Integrating temporal

planning with these techniques provides a more flexible planner which holds no bias in sequencing activity.

This system is implemented for the domain of maintenance and repair of Space Station Freedom. It currently generates VAL II instructions to a PUMA-560 with integrated force/torque and vision sensors, English instructions for a crewmember, and software procedure calls to a robotic simulator. Several extensions to this system are planned. Planning in complex domains often requires that plans be initially generated from incomplete data, or data that will evolve over time. Current methods allow the use of temporal relationships in a deductive fashion, reducing the possibilities of task ordering as more information is known about the plan. However, retracting assertions which have reduced the temporal network is computationally very expensive. Therefore, future work will concentrate on adding nonmonotonicity to the temporal logic to facilitate reasoning with changing data. This will provide a foundation to examine replanning strategies for the temporal planner. For better integration with crewmembers, techniques for explaining planner rationale will be explored. In addition, crew skill models will be developed to allow better presentation of plans and their explanations to crewmembers. These enhancements to agent modeling will also be extended to include a more sophisticated robotic agent with a dexterous three-fingered hand and advanced sensing capabilities. All of these activities will support the development of technology which allows manned spacecraft workload to be modeled and shared between crewmembers and robots, and activity plans to be automatically generated regardless of the agents who will accomplish them.

References

[1] Allen, J.F., *Maintaining knowledge about temporal intervals*, Communications of the ACM, vol. 26, 1983, pp. 832-843.

[2] Allen, J.F. and Koomen, J.A., *Planning using a temporal world model*, Proceedings IJCAI-83, 1983, pp. 741-747.

[3] Chapman, D., *Planning for Conjunctive Goals*, Artificial Intelligence, vol. 32, 1987, pp. 333-377.

[4] Hogge, J.C., *Compiling plan operators from domains expressed in Qualitative Process Theory*, Proceedings AAAI-87, 1987, pp. 229-233.

[5] Ladkin, P., *Time representation: A Taxonomy of Interval Relations*, Proceedings AAAI-86, 1986, pp. 360-366.

[6] Leban, B., McDonald, D., and Forster, D., *A Representation for Collections of Temporal Intervals*, Proceedings AAAI-86, 1986, pp. 367-371.

[7] Pelavin, R.N. and Allen J.F., *A model for concurrent actions having temporal extent*, Proceedings AAAI-87, 1987, pp. 246-250.

[8] Tenenberg, J., *Planning with abstraction*, Proceedings AAAI-86, 1986, pp. 76-80.

[9] Tsang, E.P.K., *TLP - A Temporal Planner*, Advances in Artificial Intelligence, eds. Hallam and Mellish, 1987, pp. 63-78.

[10] Vilian, M. and Kautz, H., *Constraint Propagation Algorithms for Temporal Reasoning*, Proceedings AAAI-86, 1986, pp. 377-382.

[11] Wilkins, D., Practical Planning, Morgan Kaufmann Publishers, 1988.