Hap
A Reactive, Adaptive Architecture for Agents

A. Bryan Loyall Joseph Bates

June 24, 1991
CMU-CS-91-147

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

The Hap reactive agent architecture provides many of the same mechanisms to authors as other goal-directed reactive architectures: arbitrary demons, multiple active goals and situated, runtime interpretation of plans. It also provides three unique features: convenient mechanisms for taking advantage of opportunities as they arise and for changing the agent's course of action when unfortunate events dictate; explicit mechanisms for allowing easy extension of the plan library; and flexible mechanisms for determining the success of goals without necessarily requiring the author to explicitly characterize the criteria for success. Hap was designed as part of the Oz project which is concerned with developing a rich framework for both interactive fiction and virtual realities. Oz is intended to provide a human user with the experience of living in a dramatically interesting simulated world which includes simulated intelligent and emotional agents.

This work was supported in part by Fujitsu Laboratories, Ltd.

Keywords: autonomous agents, reactive agents, artificial intelligence, interactive fiction, virtual reality, Oz

1. Introduction

Hap was designed as part of the Oz project at Carnegie Mellon. Oz is intended to provide a human user with the experience of living in a dramatically interesting simulated world which includes simulated intelligent and emotional agents. Oz provides a rich framework for both interactive fiction and virtual reality systems [5].

The Oz system architecture is shown in figure 1. The physical world contains and models all of the physical aspects of the system including the bodies of characters. These physical bodies are connected to their corresponding minds through perception/action channels. In the case of the human user(s) this channel is intercepted in each direction to allow for natural language generation [12] and parsing (including a theory of narrative presentation [16]). The drama manager can affect all of these aspects to guide the human user's experience in accordance with the author's specification.

Hap is an architecture for the minds of computer agents in Oz and is the reactive component of Tok [7], a broad architecture for agents. Other components of Tok include: a theory of emotion [14], a natural language parser and generator [12], a model of memory and a component for visualization (forward-driven simulation).

The goal of the Hap architecture is to allow for robust, reactive behavior as provided by existing hard-wired reactive architectures, such as those created by Brooks and Agre and Chapman [8, 1], while capturing a richer range of high-level actions. An explicit representation of goals is also desired, both because we believe goals are useful to organize rich behaviors, and because we wish to use the goals as data for other components of the Tok architecture (e.g. natural language and emotion).

The rest of this paper consists of four sections. After we describe the domain in which Hap agents exist, we summarize the main features of the architecture, followed by a detailed description of the architecture. We conclude by briefly comparing to other approaches and discussing the architecture's features as they relate to broad characters in dramatic story worlds and broad agents in general.

2. Domain

Hap controls characters in the Oz physical world. The world provides an object oriented simulation in which various characters, both human and computer, can interact. The simulation engine was designed to provide worlds rich enough to explore interactions between characters and create interactive stories. It contains many of the difficulties of the real world: it is complex, unpredictable and provides incomplete information. But it does not support computationally difficult details of the real world which we deem unimportant for many interesting story worlds: for example, it only supports a very simple and large-grained notion of both geometry and physics.

The world connects to the minds of characters through perception/action channels [17]. When a character wishes to perform an act it sends an act frame to its body in the world where lisp methods execute the action. An action method typically alters the world appropriately,

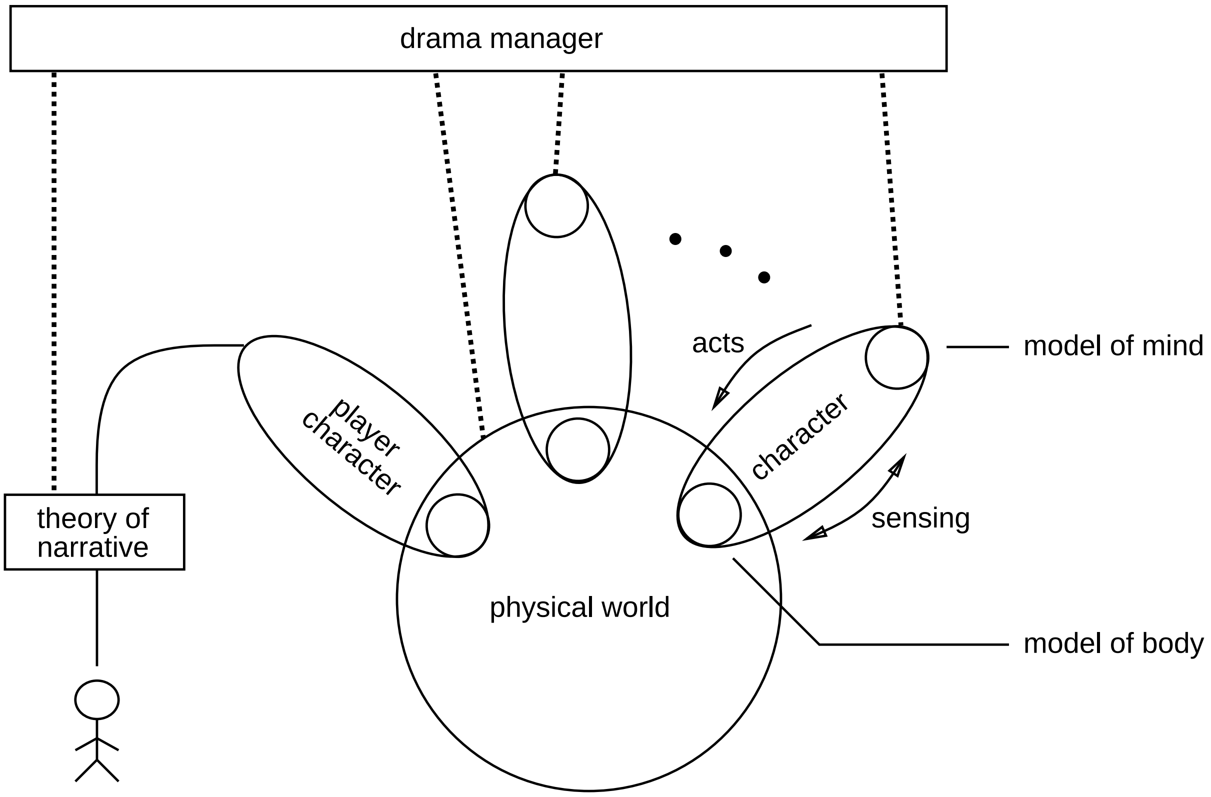

Figure 1: Oz system architecture

propagates sensory information and returns success or some kind of failure for the attempted action. If no method exists for the attempted action, sensory information is nevertheless propagated that the action was attempted and the agent's mind is notified that the action failed because no appropriate method exists.

Characters sense objects and events in the world through *sense data*. A sense datum is a collection of attribute-value pairs. Examples of attributes are size, color, class, text of speech, etc. The accuracy of the sense data varies depending on the particular sense used to perceive it, the distance it travels in the world, the objects it travels through, and the sense datum itself. For example, color cannot be determined by touch or smell, whispering can not be heard in the other room, a white shirt seen through a red glass would have a color of red, and a secret door may be seen as a normal bookcase until it is opened.

Sense data are either active or passive in the Oz world. Active sense data is intended to model events of a sufficient intensity and intrusiveness that they will be sensed (but not necessarily processed) by the agent. A siren, a gunshot or a person speaking would normally all be modeled as active in an Oz world. In contrast, passive sense data is intended to model events which in order to sense require effort on the part of the agent. The color of a shirt across the room or whether keys are on the table would both normally be modeled as passive sense data in an Oz world. Passive sense data is acquired by querying the world. Usually, most sound and smell sense data is active, while sight and touch are passive. This boundary between active and passive sense data is not fixed. In writing one story the ticking of a clock could be obtrusive, and therefore modeled as active sense data, while in another it could be background noise, and thus passive.

3. Features of the Hap architecture

In the Hap architecture, five features are important for accomplishing our goals of a rich range of robust, reactive behaviors with explicit representation of goals: (1) convenient mechanisms to allow taking advantage of opportunities as they arise and changing the agent's course of action when unfortunate events dictate; (2) multiple active goals and multiple plans for each goal; (3) runtime interpretation of plans with possible backtracking in the world; (4) an easily extended plan library; and (5) the criteria for success of goals does not have to be characterized explicitly by a predicate calculus expression. Each of these is described in turn below.

To illustrate a Hap agent's reaction to opportunities and unfortunate events, consider the following situation.

> An agent has the goal of opening a locked door. She has two applicable plans: get a key from her purse, unlock the door, and open it; or knock and wait. If, while looking for a key, someone opens the door for her, she should notice that her goal was satisfied and not keep working to accomplish it. In the same scenario, if she were searching in her purse for the key when her mischievous nephew snatched the purse, she should abandon that plan and try knocking (and perhaps deal with the nephew later.)

This example illustrates the two types of reactivity which Hap explicitly supports: recognizing the spontaneous achievement of a goal or subgoal, and realizing when changes in the world make the pursuit of an active plan nonsensical. These two mechanisms are sufficient to encode arbitrary demons as well.

Hap supports multiple active goals and multiple plans for each goal. The architecture automatically chooses which goal to pursue and how to pursue it by examining the most current perceived state of the world and the internal state of the agent. In particular, high-level goals are expanded to primitive acts incrementally at execution-time by choosing appropriate plans and their component subgoals. This runtime selection of plans allows the expression of situated plans–that is, plans whose interpretation depends on the situation in which they are executed. In this manner we capture some of the notions of situated activity and plans-as-communication as described by Agre and Chapman [2]. Plans in this library are sketchy and not perfect; they are not required always to succeed. The architecture chooses whichever plan seems best to accomplish the desired goal, and in the case of failure, tries other plans. This can result in a depth-first backtracking search of sorts. This search is different from classical depth-first search in that it takes place in the real world, so true backtracking is not possible, and whenever a branch is abandoned the new branch to search is chosen from all plans which are applicable at that time. This set of plans may be different than the set available previously. Currently, to avoid looping in this search, once a plan has failed it will not be attempted again.

The plan library can be easily extended. New plans can be added at any time without changing existing plans. To maintain reasonable behavior as new plans are added, the notion of plan *specificity* was added to each plan. The specificity of each plan is represented by a number. These numbers give a partial ordering to the plans indicating how specific a given plan is relative to another. In some cases, we intend for an author or learning component to refine existing plans by creating additional plans which apply in more specialized situations. These new plans would be given a higher specificity than the original plan. Then, if given a choice between two applicable plans, the Hap architecture will choose the more specific one, as it should be more tailored to the given situation.

The final architectural feature arose from our belief that it is difficult to create an accurate domain model of the world. To illustrate this difficulty consider this example:

> Peter has the goal to communicate to his friend Jim in another city. He tries telephoning, but gets no answer. He then uses a plan to write and send a letter. He finishes the plan by putting his letter in a mailbox, and goes on with his day.

In this example, how does Peter know that his goal was satisfied? He could wait until he finds out definitively that his message was received (by way of a reply or return receipt from the post office, for example). As a more reasonable if less correct approximation, he could have some model about the reliability of the mail system: that a message written in a letter, and placed in an envelope which has been addressed to the appropriate person satisfies the goal of conveying the message to that person. But this expression is simply a reiteration of the plan and the fact that each step was successfully finished. So, rather than require that the criterion for each goal's success be expressed in a predicate logic expression, we allow the successful completion of a plan for a particular goal be the criteria for success. The traditional approach, that a goal's success is determined by the truth or falsity of a predicate, can be realized by writing that predicate as an explicit plan success test in the last step of the plan.

4. The Hap Architecture

The Hap architecture is shown in figure 2. To illustrate the architecture and the above features, let's examine our agent who wants to get through the locked door. At any specific time the active goals and plans for a Hap agent are recorded in the *active plan tree* of the agent, and all of the plans known to the agent are stored in *plan memory*. So, for our example agent, the goal to open the door is in the active plan tree along with other current goals and plans, and the two plans for opening a door are in plan memory.

The active plan tree is interpreted in a three step loop. First, the tree is adjusted for changes in the world by removing nonsensical plans and spontaneously achieved goals. Next one of the goals in the tree is chosen to execute. In the third step, the chosen goal is executed. If this goal is a primitive action then it is executed by simply doing it. If it is a subgoal then it is used, along with the current world state, to index plan memory and choose a plan. This plan and its component subgoals are then placed in the tree for consideration in the subsequent passes through the loop.

In the first pass through the loop for our example agent, the plan to open the door is chosen to be executed, and the plan to unlock the door with a key from her purse is chosen

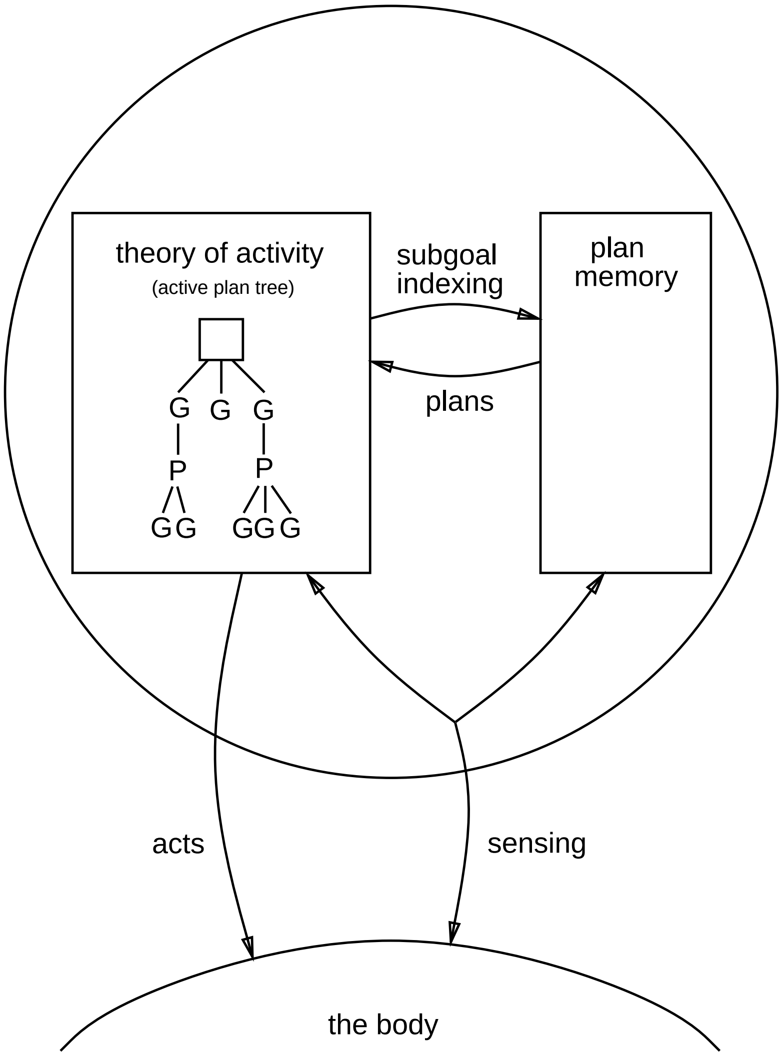

Figure 2: The Hap Architecture

as a method of executing it. This plan and its component subgoals are then added to the active plan tree. On subsequent passes through the loop, these subgoals could be chosen to be executed. If, during these passes, our agent no longer has her purse and has not yet gotten the key from it, the plan would no longer make sense, and would be removed during the first step of the loop. Likewise, if the door is opened, work on the original goal would no longer be necessary, and so the goal would be removed. If neither of these happen, the subgoals would be further expanded using other applicable plans from plan memory, until a sequence of primitive actions accomplishes the goal of opening the door.

4.1. Active Plan Tree

Figure 3 shows a more detailed instance of an active plan tree from our example agent. The active plan tree is composed of alternating layers of goals and plans. Each plan node in the tree has some number of goal nodes as its children. These goal nodes are the subgoals which must be satisfied for the plan to succeed. Each goal node either has no children (and thus it is a leaf goal node), or its child is the current plan for the goal. The tree grows as plans are chosen for leaf goals and shrinks as plans and goals succeed and fail.

A plan node accomplishes its purpose (it succeeds) once all of its children succeed. It fails if there is no way for one of its subgoals to succeed. If desired by the author, a plan can have an explicit plan success test which must also evaluate to true for the plan to succeed. This test is evaluated after the plan has finished. A plan node can be either sequential or parallel. The subgoals of a sequential plan must be accomplished in order, while the subgoals of a parallel plan may be accomplished in any order.

There are several types of goal nodes. A subgoal goal node succeeds if a successful plan can be found for it in the current context. A primitive mental action always succeeds. A primitive body action goal node succeeds if the action is successful in the world. This notion of success is very simplistic, so a separate goal node type is provided for primitive actions which ignores the success or failure information from the world. This goal node always succeeds when it executes and extra processing may be done by the enclosing plan to determine actual success or failure.

At the root of the tree is a special node which is parent to all of the top-level goals of the agent. These top-level goals are classified according to whether they are persistent. The persistent goals are continuous goals and thus are never removed from the tree. Other goals are removed from the tree once they have succeeded or failed.

Associated with each goal node is a *success-test* and a *priority*. The success-test is a method for recognizing when a goal's purpose in the enclosing plan is spontaneously achieved (capturing the first type of reactivity above.) It is an expression about the state of the world. If it ever becomes true, then this goal node has succeeded and is thus removed from the tree. The subtree rooted at the goal node is, in the process, also removed, since its only purpose was working toward the goal. Success-tests are sufficient, but not necessary for a goal to succeed. A very unreactive agent might use *false* for most success-tests and therefore never recognize when goals are spontaneously achieved. The priority is a number representing how important the goal is to the agent. Preferring higher priority goals is one of the methods used by the architecture to arbitrate between multiple goals.

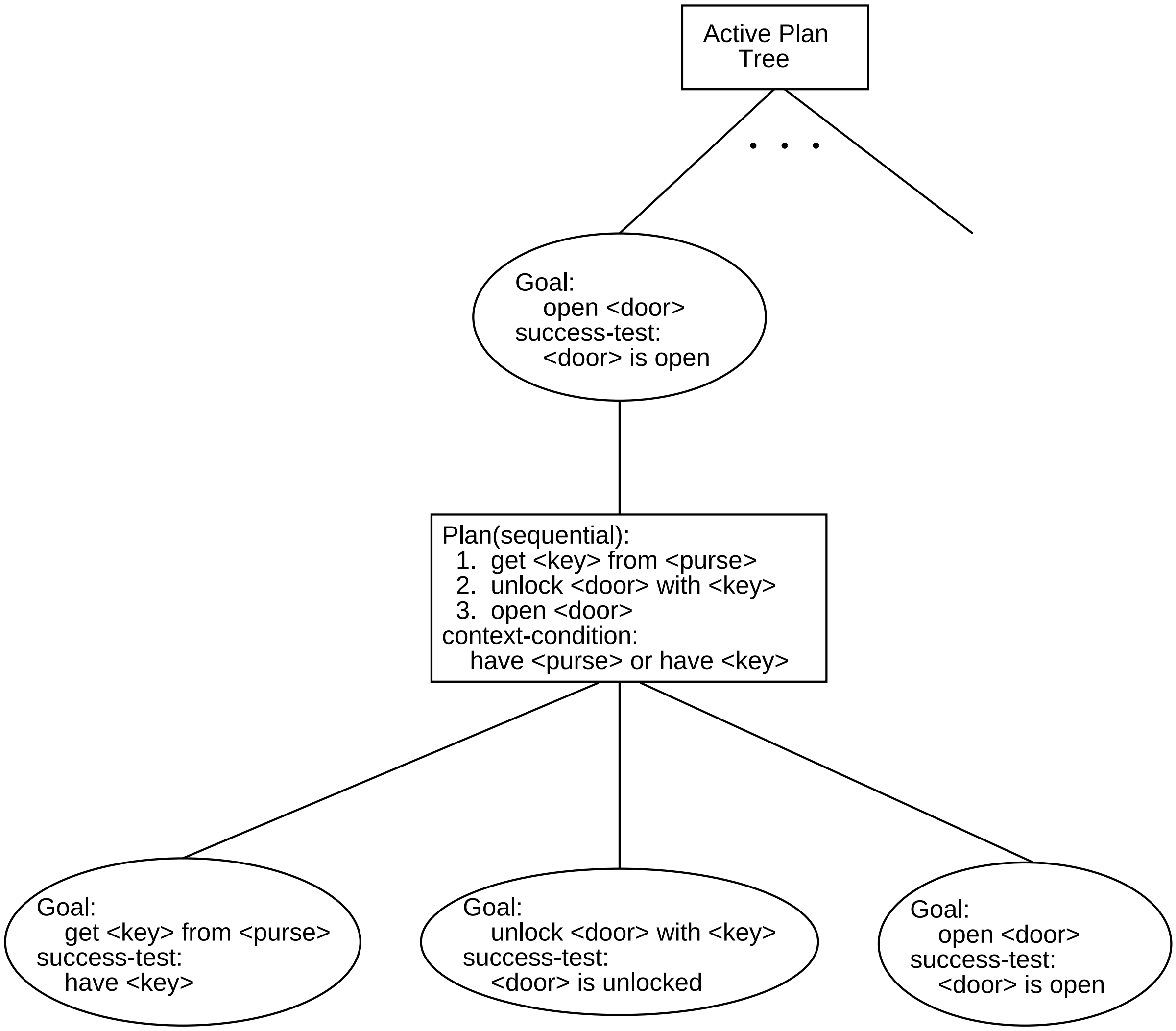

Figure 3: An example active plan tree

Associated with each plan node is a *context-condition* and a *specificity*. It is intended that the context-condition must continuously yield true for the plan to have any chance of making sense and achieving its goal. Thus the context-condition is necessary, but not sufficient, for the plan to work. A very dumb character might use *true* for most context-conditions. This corresponds to not being very aware of the world changing out from under a previously plausible plan. Specificity is a measure of how specific each plan is. More specific plans are preferred over less specific plans by the plan arbiter (see the theory of activity below).

Both success-tests and context-conditions are demonstrated in the example active plan tree of figure 3. The goal node to open the door has a success-test that checks to see if the door has been opened for her. And the plan has a context-condition which checks to make sure that she is still holding her purse (or has the key already in hand).

4.2. Plan Memory

Plan memory is a set of production rules. The left hand side of each production is composed of a goal-expression and a precondition-expression. The goal-expression contains the goal name and zero or more variables. For a subgoal to match a production, the goal names must match and the number of values given in the subgoal must match the number of variables in the goal-expression. In addition, the precondition-expression must be true in the current world state for the production to be applicable. The precondition-expression can reference the values from the goal-expression variables in its tests.

The right hand side of the production contains the context-condition, specificity and plan expression which are instantiated to create the plan and subgoal nodes for the active plan tree. The plan node is created using all three of these, and the plan expression is used to create the subgoal nodes. The plan expression is either a parallel or sequential arrangement of steps. Each step contains, a goal expression, a priority modifier and a success-test expression. The goal expression and success-test expression are used directly to create the appropriate goal node, and the priority modifier is added to the priority of the parent goal node to create the priority of this node. In this way a subgoal can be more, less or the same level of importance as the goal node which spawned it.

4.3. Theory of Activity

The execution loop consists of three steps: adjust the active plan tree based on changes in the world, pick a leaf goal to execute, and execute the goal. Executing the goal can take the form of performing a primitive physical or mental act, or expanding a subgoal.

The first step of the execution loop is to adjust the active plan tree based on changes in the perceived world state. The entire active plan tree is examined at this time, and both plans which no longer apply in the current context and (sub)goals that have already been satisfied are removed. Plans are detected as no longer applying if the associated context-condition evaluates to false. Goals are detected as having already been satisfied if the associated success-test evaluates to true. In either case, all attempts to use the plan or to satisfy the goal are aborted by removing the subtree rooted at that node from the active plan tree. The parent to the removed node is notified that either the goal has been achieved or that the plan has failed. This effect can then propagate up the tree causing enclosing plans and goals to either succeed or fail. After the tree is adjusted, computation continues with the rest of the tree. In the normal process of this computation, one of the failed plans may be replaced by a new applicable plan resulting in a search through the world as described previously. If no applicable plan can be found the associated goal will fail.

In the next step, the next leaf goal to execute is chosen by the *goal arbiter*. Currently it chooses a leaf goal in the following manner. First, it prefers higher priority goals. If there is still ambiguity in which leaf goal to execute, it prefers to work on the same higher-level goals that it was last working on rather than switching to new higher-level goals. Finally it will randomly choose among remaining leaf goals. In the future, the goal arbiter may be extended to consider deadlines of goals, the ease of accomplishing a goal, or the potential for accomplishing multiple goals.

Once a goal has been chosen, it is executed. Each type of goal node has its own method of execution. A primitive physical action is executed by sending the action frame to the body of the agent. A primitive mental action is executed by simply performing it. (This actually takes the form of evaluating a lisp expression, and thus can affect any part of the Hap mental state.) A subgoal is executed by using the subgoal expression along with the perceived world state to index plan memory. From the resulting set of applicable plans one is chosen to be attempted. A plan node and subgoal nodes are created by instantiating the chosen plan, and the created nodes are placed in the active plan tree for consideration the next time through the loop.

Choosing which plan to attempt is the job of the *plan arbiter*. Currently it picks one of the matching plans by first preferring more specific plans to less specific plans, and then choosing randomly. Future extensions to the plan arbiter might include using statistics of the success rate of plans and expected cost of plans to make better decisions.

After executing the chosen goal node as appropriate, the theory of activity continues with the first step of the loop.

5. Discussion

Several systems have been created in the past few years with the intent of combing reactivity with rich goal-directed behaviors. Reactive Action Packages (RAPs) by Firby [9], the Procedural Reasoning System (PRS) by Georgeff, et al [10, 11], and the Task Control Architecture (TCA) by Simmons [13, 15] are all particularly similar to Hap in that they provide an explicit representation of goals, provide mechanisms to respond quickly to emergencies, and provide the capability to handle multiple active goals. Hap differs from these previous approaches on three points: (1) Hap provides explicit, convenient mechanisms for recognizing when a goal has been spontaneously achieved or when a course of action has become invalid; (2) Hap's plan library was designed to be incrementally extended; and (3) in Hap the success condition of a goal does not have to be characterized explicitly in order for the goal to succeed.

As agents become broader and longer lived, we believe each of these capabilities will become increasingly important. Certainly for our purpose–creating characters to live in dramatic story worlds–each of these is important. Characters in dramatic story worlds must have broad behavior if they are to avoid destroying the human participant's suspension of disbelief. It is therefore important that these characters recognize and react well to both opportunities and events which alter the appropriateness of the agent's present course of action. With the explicit, convenient mechanisms provided by Hap for recognizing when a goal has been achieved or when a plan has become invalid, as well as the standard arbitrary demons provided by all of these architectures, we expect it to be easier for authors to encode this level of reactivity.

Characters in dramatic story worlds and, we believe, broad agents in the real world, must express many behaviors with the property displayed by our example agent who wrote a letter to his relative. As in the case of sending mail, the world often does not provide immediate feedback with which to determine success or failure of an agent's plans. For this reason, the capability to use successful completion of a plan as the criteria of success for the plan is useful.

We expect characters in Oz and broad agents to have a long life, and behaviors from past characters to be reused when creating new characters. As behaviors are used over a period of time, it is useful for an author or learning component to be able to modify them incrementally and locally. Hap's mechanism for incrementally extending the plan library is designed for exactly this.

The Hap system as described has been implemented and several small agents have been built using it. Further work will proceed along two paths: building larger agents in the Hap architecture itself, and integrating Hap with the rest of Tok.

References

[1] Philip E. Agre and David Chapman. Pengi: an implementation of a theory of activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, Seattle, WA, July 1987. AAAI.

[2] Philip E. Agre and David Chapman. What are plans for? In *Robotics and Autonomous Systems*, pages 17–34. Elsevier Science Publishers, 1990.

[3] Joseph Bates. Computational drama in Oz. In *Working Notes of the AAAI-90 Workshop on Interactive Fiction and Synthetic Realities*, Boston, MA, July 1990.

[4] Joseph Bates, editor. *Working Notes of the AAAI-90 Workshop on Interactive Fiction and Synthetic Realities*, Boston, MA, July 1990. American Association of Artificial Intelligence.

[5] Joseph Bates. Deep structure for virtual reality. Technical Report CMU-CS-91-133, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1991. Forthcoming in *Presence*, The Journal of Teleoperators and Virtual Environments, MIT Press, 1991.

[6] Joseph Bates, A. Bryan Loyall, and W. Scott Reilly. Broad agents. In *Proceedings of AAAI Spring Symposium on Integrated Intelligent Architectures*, Stanford, CA, March 1991. Available as a special issues of SIGART News, Summer 1991.

[7] Joseph Bates, A. Bryan Loyall, W. Scott Reilly, and Mark Kantrowitz. Tok: A broad architecture for intelligent, emotional agents. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, forthcoming Summer 1991.

[8] Rodney Brooks. Intelligence without representation. In *Proceedings of the Workshop on the Foundations of Artificial Intelligence*, June 1987.

[9] R. James Firby. *Adaptive Execution in Complex Dynamic Worlds*. PhD thesis, Department of Computer Science, Yale University, 1989.

[10] Michael P. Georgeff and Francois Felix Ingrand. Decision-making in an embedded reasoning system. In *the Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, MI, August 1989.

[11] Michael P. Georgeff, Amy L. Lansky, and Marcel J. Schoppers. Reasoning and planning in dynamic domains: An experiment with a mobile robot. Technical Report Technical Note 380, Artificial Intelligence Center, SRI International, Menlo Park, CA, 1987.

[12] Mark Kantrowitz. Glinda: Natural language text generation in the Oz interactive fiction project. Technical Report CMU-CS-90-158, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1990.

[13] Long-Ji Lin, Reid Simmons, and Christopher Fedor. Experience with a task control architecture for mobile robots. Technical Report CMU-RI-TR-89-29, Robotics Institute, Carnegie Mellon University, 1989.

[14] W. Scott Reilly and Joseph Bates. Em: Proposal for a computational model of emotions in Oz. Working Paper, Oz Project, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA., May 1991.

[15] Reid Simmons. Concurrent planning and execution for a walking robot. In *Proceedings of the IEEE International Conference on Robotics and Automation*, Sacremento, CA, April 1991.

[16] Sean Smith and Joseph Bates. Toward a theory of narrative for interactive fiction. Technical Report CMU-CS-89-121, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1989.

[17] Jon Swartz and Joseph Bates. The mind-body interface in Oz. Working Paper, Oz Project, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA., May 1991.