JAM: A BDI-theoretic Mobile Agent Architecture

Marcus J. Huber
Intelligent Reasoning Systems
4976 Lassen Drive
Oceanside, CA 92056
marcush@home.com

ABSTRACT

JAM is a hybrid intelligent agent architecture that draws upon the theories and ideas of the Procedural Reasoning System (PRS), Structured Circuit Semantics (SCS), and Act plan interlingua. Furthermore, JAM draws upon the implementation pragmatics of the University of Michigan's and SRI International's implementation of PRS (UMPRS and PRS-CL, respectively). JAM provides rich and extensive plan and procedural representations, metalevel and utility-based reasoning over multiple simultaneous goals, and goal-driven and event-driven behavior that are an amalgam of all of the sources listed above. The JAM agent architecture also provides an agentGo primitive function utilizing Java's object serialization mechanism to provide widely-supported mobility capabilities.

Keywords

Agent architecture, intelligent agent, belief-desire-intention architecture, procedural reasoning system, intelligent software agent.

1. INTRODUCTION

We developed the JAM intelligent agent architecture as the next step in the evolution of pragmatic BDI-based agent architectures. Our approach to agent architectural theories and design has been to start with a BDI-theoretic core followed by incremental improvements based upon incorporation or modification based on the research of others and our own ongoing research and experience. JAM combines what we believe to be the best aspects of several leading-edge intelligent agent frameworks, including the original BDI (belief, desire, intention) theories and specification of the Procedural Reasoning System (PRS) of Georgeff, Lansky, Rao, and others [6, 10], the Structured Circuit Semantics (SCS) representation of Lee and Durfee [13], and the Act plan interlingua [25, 16] of Myers, Wilkins and others. In addition, JAM draws upon pragmatics garnered from the PRS implementations of the University of Michigan (called UMPRS) [9, 13]) and SRI International (called PRS-CL) [17].

Starting with a BDI-theoretic "kernel" allows us to reap the benefits of a large body of research on the theory and implementation of, in particular, the Procedural Reasoning System (PRS). Explicit modeling of the concepts of *beliefs*, *goals* (desires), and *intentions* within an agent architecture provides a number of advantages, including facilitating use of declarative representations for each of these concepts. The use of declarative representations in turn facilitates automated generation, manipulation, and even communication of these representations. Other advantages of starting with a PRS-based BDI architecture include a sound model of goal-driven and data-driven task achievement, metalevel reasoning, and procedurally-specified behavior.[1] And although capabilities are not a central attribute of BDI theories, implemented architectures typically include explicit, declarative modeling of the capabilities of the agent in the form of plans and primitive actions.

Alternative agent architectures provide some, but not all of the advantages of the BDI-based PRS framework. Many "agent" architectures such as Aglets and Java Agent Template (JAT) from IBM, Agent Tcl [7] from Dartmouth, and Agents for Remote Action (ARA) from the University of Kaiserslautern have little or no explicit representations of any of the mentalistic attributes of beliefs, intentions, capabilities, or even goals (often argued to be the key feature of agency). Furthermore, programmers encode the behavior of these agents almost completely through low-level hardcoding. Each of these agents provides specialized functionality in some focus area (e.g., ARA and Agent Tcl are specialized to provide mobility capabilities) but do not otherwise provide what we consider a complete reasoning architecture.

Agent architectures such as SOAR [12] and AOP-based (Agent-Oriented Programming) [21] architectures such as Agent-0 [22], LALO, and PLACA [23] all provide significantly more complete representation and reasoning frameworks than those mentioned in the preceding paragraph. Soar implements a unified theory of cognition [18] and provides a wide range of desired agent architecture capabilities, including integrated execution, means-ends planning, metalevel reasoning, and learning. The primary disadvantage of Soar is its highly unintuitive and constrained rule-based behavior representation and reasoning cycle. It would require a significant overhaul of Soar to improve the procedural expressiveness of its rule representation. The AOP-based architectures provide explicit internal representations of mentalistic concepts such as beliefs and commitments but they emphasize social interaction capabilities over individual capabilities (even though PLACA does extend the AOP paradigm to include generative planning capabilities).

[1] Note that the *possible worlds* model upon which PRS is formally based [19, 20] is only implicitly represented within any pragmatic implemented BDI architectures.

In addition to these primarily monolithic agent architectures are a number of multi-level agent architectures such as TouringMachines [3], Atlantis [5], and InteRRap [15]. Agent architectures of this style vary widely in their theoretical foundation, internal representations, architectural components, and particular emphasis on specific representational or behavioral issues or application domain. One common problem with multi-layer architectures is that they require a specialized programming language for each layer (e.g., reactive layer language, scheduling layer language, planning layer language, coordination layer language). None of these architectures provides as mature or cohesive a theoretical basis as that provided by the BDI theories and PRS specification. Both the BDI theories and PRS specification have their limitations too, of course, but we believe they provide a much stronger starting point.

Our research began with the development of UMPRS starting in 1992 to provide us with a well-founded means of encoding behavior for mobile robots [13] for performing mobile robot research and since has been used for a number of additional application areas [2, 8, 11, 24]. UMPRS implements a majority of the representational and behavioral concepts found in the original PRS specification and was extended to provide a superior, more structured procedural representation. UMPRS lacks a strong architectural conceptualization of goals (not suffered by PRS or PRS-CL) however, and lacks some significant procedural constructs (e.g., parallel execution) and goal types (e.g., homeostatic goals). JAM does not suffer from these shortcomings, and provides strong goal-achievement syntax and semantics with support for homeostatic goals and a much richer, more expressive set of procedural constructs.

The stronger, formalized goal implementation of JAM is motivated by work in progress on extensions to support generative planning capabilities within the JAM architecture and originate in the original PRS specification and the large body of classic AI planning research. Procedural extensions were garnered from the Structured Circuit Semantics (SCS) representation [14] and the Act plan interlingua [16, 25]. Functional improvements such as agent mobility are founded in the research from the Dartmouth and IBM mobile agent research efforts described earlier.

JAM has already been used in a number of projects including Johns Hopkins University's Applied Physics Laboratory and Orincon Corporation. JHU/APL implemented a JAM agent as a plan execution monitoring agent for robotic navigation and also for chemical weapons attack detection and recovery advice. Orincon Corporation is extending its Agent Workbench toolkit to build JAM agents in addition to UMPRS agents.

In the remainder of the paper, we describe the JAM architectural components, representations, and salient functionality and future work.

2. JAM

Each JAM agent is composed of five primary components: a *world model*, a *plan library*, an *interpreter*, an *intention structure*, and an *observer*. We illustrate this in Figure 1. The world model is a database that represents the beliefs of the agent. The plan library is a collection of plans that the agent can use to achieve its goals. The interpreter is the agent's "brains" that reason about what the agent should do and when and how to do it. The intention structure is an internal model of the agent's current goals and keeps track of the commitment to, and progress on, accomplishment of those goals. The observer is a user-specified lightweight declarative procedure that the agent interleaves between plan steps (in addition to the reasoning performed by the JAM interpreter) in order to perform functionality outside of the scope of JAM's normal goal/plan-based reasoning (e.g., to buffer incoming messages).

The JAM execution semantics and behavior is a combination of that of UMPRS and SCS. Changes to the world model or posting of new goals triggers reasoning to search for plans that might be applied to the current situation. The JAM interpreter selects one plan from this list of applicable plans based on either metalevel reasoning or maximum utility, *intends* it (i.e., commits itself to execution of the instantiated plan), and executes the first intention found with the highest utility (i.e., as if there was an SCS DO_BEST over all top-level goals). The remainder of this section discusses each of the major components of JAM in as much detail as space permits.[2]

2.1 INTERPRETER

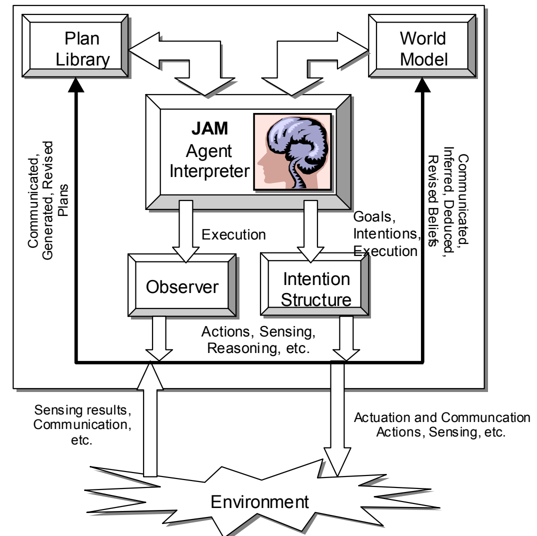

Figure 1. The JAM intelligent agent architecture.

The JAM interpreter is responsible for selecting and executing plans based upon the intentions, plans, goals, and beliefs about the current situation. Associated with the interpreter is the *intention structure*, a run-time stack (stacked based upon subgoaling) of goals with and without instantiated plans. A JAM agent may have a possibly large number of alternative plans for accomplishing any single goal and the JAM interpreter reasons about all of the alternatives combinations of plans, goals, and variable bindings (based on goal arguments and beliefs) before selecting the best alternative given the particular situation. The JAM interpreter's top-level loop is shown in Figure 2.

The agent checks all the plans that can be applied to a goal to make sure they are relevant to the current situation. Those plans

[2] Complete details on JAM can be found at http://members.home.net/marcush/IRS/Jam/jam-man.doc.

that are applicable are collected into what is called the Applicable Plan List (or APL). A utility value is determined for each instantiated plan in the APL and, if no metalevel plans are available to select between the APL elements, the JAM interpreter selects the highest utility instantiated plan (called an *intention*) and *intends* it to the goal. Note that neither the original PRS specification nor prior PRS-based implementations (such as PRC-CL) supports utility-based reasoning. Also, in contrast to the JAM interpreter, the UMPRS interpreter does not actually intend an intention if a new intention does not have a utility higher than all of the intentions already on the agent's intention structure. Continuing with JAM's interpreter behavior, if the goal with the new intention has the highest utility among all goals with intentions, then the new goal's plan is executed. Otherwise, a previous intention still has a higher utility and the interpreter executes that intention's plan.

```
public int run() {
  // loop forever until agent completes all of its goals
  while (true) { // outer, infinite loop
    // execute the Observer procedure
    returnValue = intentionStructure.executePlan(observer);
    metaLevel = 0;
    while (true) { // metalevel loop
      // generate an Applicable Plan List (APL) if necessary
      apl = new APL(planLibrary, worldModel,
                    intentionStructure, metaLevel);
      // If the new or previous APL is not empty then add
      // entry to the World Model to trigger the next level of
      // reasoning.
      if (apl.getSize() != 0)
        worldModel().assert("APL", metalevel, apl, aplSize);
      // If this APL is empty then no new level of reasoning
      if (apl.getSize() == 0) {
        if ((intentionStructure.allGoalsDone()))
          return 0;
        // If the previous APL was empty then execute
        // something in the intention structure, otherwise
        // select something from the previous APL, intend it,
        // and then run something in the intention structure.
        if (last_apl == null || last_apl.getSize() == 0) {
          intentionStructure.run();
          break;
        }
        else {
          selectedElement = last_apl.getHighestRandomUtility();
          intentionStructure.intend(selectedElement);
          intentionStructure.run();
          last_apl = null;
          break;
        }
      }
      else {
        last_apl = apl;
        metaLevel++;
      }
    } // Inner, metalevel loop

    // Clear the World Model of APL elements.
    getWorldModel().retract("APL");

  } // Outer infinite loop
}
```

Figure 2. The JAM agent's main interpreter loop.

If generation of an APL results in one or more entries, the agent possibly enters into *metalevel* reasoning. That is, it reasons about how to decide which of the APL elements to intend to its intention structure. The agent may have multiple ways (i.e., applicable metalevel plans) of performing this decision-making, so that metalevel reasoning about its metalevel reasoning may ensue. Metalevel reasoning ends when the interpreter no longer generates a non-null APL, indicating that the agent has no higher metalevel means for deciding between alternatives. The JAM distribution comes with a number of implemented primitive actions to facilitate metalevel reasoning. These include primitives for selecting an intention from an APL, finding the goal currently being pursued, finding the plan currently being executed, extracting a plan's attributes and attribute values, and intending an intention onto the Intention Structure.

2.2 GOALS

A JAM agent's top-down behavior is motivated by specifying top-level goals. Goals can be given to the agent in a text form with the following syntax:

```
goal_type goal_name parameter1 ...  parameterN
                       <:UTILITY expression>;
```

The goal type is one of: ACHIEVE, PERFORM, and MAINTAIN. The goal name is a label identifying the goal relation, and the parameters are the goal relation's arguments. The :UTILITY keyword and `expression` are optional and provide an opportunity to specify either a fixed numeric value (which then cannot be regarded strictly as a utility but we permit such use) or an arbitrarily complex utility calculation (perhaps involving probabilistic information). As we will describe in more detail later, the goal's utility is combined with the utility of an instantiated plan to calculate the total utility of the intention. A JAM agent will dynamically switch between alternative goals as the intention utilities change so that it is always pursuing the highest utility intention. In the sense that JAM agents are utility maximizing, the JAM architecture results in strictly rational agents.

One or more top-level goals are initially given to the agent at agent invocation. The text specification for top-level goals are specified by using the keyword "GOALS:" and then a list of goal specifications in the form specified above. An example top-level goal specification for a robotic agent that wanders around a building's lobby greeting guests might look like:

```
GOALS:
    PERFORM wander_lobby;
    ACHIEVE initialize :UTILITY 300;
    MAINTAIN charge_level "20%";
    MAINTAIN safe_distance_from_obstacles 50.0;
```

This list of goals can be augmented during execution through communication with other agents, generated from internal reasoning on the part of the agent, or by many other means.

Top level goals are *persistent*. That is, they are pursued until they are satisfied by successful plan execution or opportunistically by some other means, such as another agent, or are removed explicitly within a plan (perhaps because the agent believes it is no longer capable of achieving the goal). If a plan for a top-level goal fails, the agent removes its commitment to that goal by removing its intention, but leaves the goal on the intention structure for later attempts at achieving the goal. With this operationalization of top-level goals, agents have a level of commitment to all top-level goals and an especially strong level of commitment to goals that have intentions associated with them (which is consistent with, for example [1] and formal BDI definitions). A further difference between subgoals and top-level goals is that subgoals are *not* persistent by default. If a plan fails for a subgoal, the interpreter considers the subgoaling action to have failed (just as if it were any other type of plan action).

JAM supports a number of different types of goals, ACHIEVE, PERFORM, and MAINTAIN, each with distinct semantics. An ACHIEVE goal specifies that the agent desires to achieve a goal state and is the goal type typically associated with BDI architectures and generative planning systems. For ACHIEVE goals, the JAM interpreter checks to see whether the goal has already been accomplished before generating an APL. If the goal has been accomplished, the agent does not actually establish a top-level goal or subgoal. JAM agents continually monitors for goal achievement. Typically, the plan selected for the subgoal will be the means by which the subgoal is accomplished. However, if the agent detects opportunistic accomplishment of the goal, perhaps by another agent, it will consider the subgoal action successful and discontinue execution of the plan. Finally, a world model entry indicating that the goal has been achieved is asserted if the plan selected for accomplishment of the ACHIEVE subgoal completes successfully. The world model entry that is asserted is the goal specification for the goal just achieved.

A PERFORM goal specifies the agent desires to perform some behavior. This semantics is an extension not found in typical BDI architectures and indicates that the agent is *not* interested in achieving a goal, per se, but merely to exhibit a particular behavior. PERFORM goals differ from ACHIEVE goals in several important aspects. The agent does not check to see whether the goal has already been accomplished before selecting plans to perform the behavior. The agent does not monitor for goal achievement during plan execution and will execute the intended plan until the plan succeeds or fails. Finally, an assertion to the world model entry that the goal has been achieved is only performed if the plan that was executed has an ACHIEVE goal specification and it completes successfully.

A MAINTAIN goal indicates that the specified goal must be reattained if it ever becomes unsatisfied. A MAINTAIN goal is similar to an ACHIEVE goal except that a MAINTAIN goal is never removed from the agent's goal list automatically (i.e., it is a *homeostatic* goal). A MAINTAIN goal can be removed from the agent's intention structure only by the agent explicitly removing it.

2.3 PLANS

A JAM plan defines a procedural specification for accomplishing a goal, reacting to an event, or performing behavior. JAM agents are therefore capable of both goal-driven and data-driven behavior. The basic structure of a JAM plan is shown in Figure 3. Optional plan fields are surrounded with the "<" and ">" symbols.[3] One or more plans are initially given to the agent at agent invocation. The text specification for plans are specified by using the keyword "PLANS:" and then a list of plan specifications in the form specified below. This list of plans can be augmented during execution through communication with other agents, generated from internal reasoning on the part of the agent, or by many other means.

A plan's applicability is limited to either a particular goal or a data-driven conclusion. Each plan may be further constrained to a particular *precondition,* conditions that must hold before starting execution of the plan, and *context,* conditions that must hold both before and during execution of the plan. This semantic differentiation of runtime and pre-runtime conditions provides more flexibility to an agent programmer than does the context semantics found in the PRS or Act specifications.

The procedure to use to accomplish the goal is given in the plan's

```
PLAN: {
    GOAL: [goal specification]
        or
    CONCLUDE: [world model relation]
    NAME: [string]
    BODY: [procedure]
    <DOCUMENTATION: [string]>
    <PRECONDITION: [expression]>
    <CONTEXT: [expression]>
    <UTILITY: [numeric expression]>
    <FAILURE: [non-subgoaling procedure]>
    <EFFECTS: [non-subgoaling procedure]>
    <ATTRIBUTES: [string]>
}
```

Figure 3. Anatomy of a JAM agent plan.

procedural *body*, which can contain simple actions (e.g., execute a user-defined primitive function) and complex structured constructs (e.g., iteration and equivalents to if-then-else). JAM (and UMPRS) use structured programming constructs in contrast with previous instantiations of PRS (e.g., PRS-CL), that allow unstructured procedures (i.e., procedures with the equivalent of "goto" actions.), which we believe to represent a significant improvement for agent programmers.

Each plan may include an explicitly or implicitly defined *utility* calculation, which is used to influence selection of certain procedures over others through the default utility-based metalevel reasoning mechanism of JAM. The utility calculation can be a fixed value or an arbitrarily complex calculation involving instantiated variables and possibly

Another optional component is the *effects* field[4], which is a procedure that the JAM interpreter executes when the plan completes successfully. An agent programmer can use the effects field to perform World Model updating, which would result in behavior similar to the add/delete list in STRIPS plans [4], but

---

[3] The full BNF grammar for JAM agents can be found at http://member.shome.net/marcush/IRS/Jam/Jam-man.doc.

[4] Note that UMPRS plans have a field with the same name but that the semantics of the UMPRS implementation is quite different. UMPRS uses the EFFECTS field in a special simulation mode and does not use it during normal execution.

can also use it to execute any other procedural construct other than subgoaling.

A procedural specification of what the agent should do when a plan fails can be represented in a plan's optional *failure* section. This is similar to the effects field in that it is a procedure that can contain any JAM plan component except subgoaling. An agent programmer can use the failure section to define "cleanup" code specific to the particular plan.

Name	A/C	Description
ACHIEVE	A	Establish goal/subgoal
AND	C	Do all branches; try in order
ASSERT	A	Add relation to world model
ASSIGN	A	Set variable value
ATOMIC	C	Perform sequence of actions without interruption
DO_ALL	C	Do all branches in random order
DO_ANY	C	Do one randomly selected branch
DO … WHILE	C	Iterate
EXECUTE	A	Perform primitive action
FACT	A	Check world model entries
FAIL	A	Unconditionally fail
LOAD	A	Parse JAM input file
MAINTAIN	A	Homeostatic goal
OR	C	Do any branch; try in order
PARALLEL	C	Execute all branches simultaneously (as permitted by hardware)
PERFORM	A	Establish behavior goal/subgoal
POST	A	Add top-level goal
QUERY	A	Establish goal/subgoal
RETRACT	A	Remove from world model
RETRIEVE	A	Get values from world model
TEST	A	Check condition
UNPOST	A	Remove goal
UPDATE	A	Change world model
WAIT	A	Wait for condition/goal
WHEN	C	Conditional execution
WHILE	C	Iterate

Table 1. Available plan actions and constructs in JAM. The middle column indicates whether the item is an action (A) or a construct (C).

The optional *attributes* plan field provides a place for a programmer to put information concerning plan characteristics that the agent can reason about during plan execution and metalevel reasoning.

The *name* and *documentation* fields are placeholders for a unique identifier string and explanatory textural documentation that should accompany the plan, respectively.

JAM provides many programming actions and constructs. We define actions to be single-line statements and constructs to be multiple-line statements. We list each of the built-in actions and constructs in Table 1.

JAM provides many standard programming constructs such as iteration (represented by DO … WHILE and WHILE constructs), conditional branching (represented in various specialized forms by OR, AND, DO_ALL, DO_ANY, and WHEN), and variable value setting (ASSIGN). A JAM agent's beliefs can be changed and checked using ASSERT, FACT, RETRACT, RETRIEVE and UPDATE. JAM provides constructs for true simultaneous parallel activity (PARALLEL) and a synchronizing construct (WAIT). The PARALLEL action provides simultaneous execution of multiple plan branches using separate Java threads while preserving the JAM execution semantics of interleaving action execution with interpreter reasoning (including Observer execution). JAM's WAIT construct causes plan execution to pause until a specified goal is achieved or a specified action returns successfully. Execution of the plan continues "in place", with the agent checking the goal or action every cycle through the interpreter.

```
PLAN: {
      NAME: "Plan 1: Gather and process information"
      GOAL: ACHIEVE information_exploited $user_query $result
            $recursed;
      BODY:
         EXECUTE com.irs.jam.primitives.GetHostname.execute
               $hostname;
         EXECUTE print "Currently at " $hostname "\n";
         OR
         { // Check to see if we are done
            TEST (querySatisfiedp $user_query $solution);
            EXECUTE print "Done working on query.\n";
         }
         { // We are not done, so figure out what to do next
            EXECUTE determineNextInfoSource $user_query
                  $nextHostname $nextPort $result;
            EXECUTE agentGo $nextHostname $nextPort;
            EXECUTE gatherAndProcessInfo $query $result;
            ACHIEVE information_exploited $user_query $result
               "true";
         };
         WHEN : TEST (== $recursed "false")
         {
               agentGo $hostname $port;
         };
}
```

Figure 4. An information gathering plan using goal-directed behavior and agent mobility.

We show an example of some simple JAM agent plans in Figure 4 and Figure 5. Figure 4 demonstrates a goal-driven (ACHIEVEment-based) plan that employs JAM's mobility functionality. The basic idea of the plan is that a parent plan establishes a subgoal with a user-based query as a parameter and the result is returned by plans achieving that goal. The plan establishes the originating computer host, determines information sources that the agent needs to gather information from, processes the new information, and recurses. When the query is satisfactorily answered, the agent returns to its original computer platform. Figure 5 demonstrates a data-driven plan that performs simple metalevel reasoning and bases its decision on lowest cost. This second plan searches through the plans in the Applicable Plan List (APL) and looks for the plan with the lowest value for "cost" in the plans' ATTRIBUTE field. The lowest-cost plan is intended if there is a single lowest-cost plan, otherwise a plan is randomly selected and intended from the APL.

```
Plan: {
NAME: "Plan 2:Metalevel reasoning"
DOCUMENTATION: "Perform metalevel reasoning"
CONCLUDE: APL $LEVEL $APL $APLSIZE;
CONTEXT: (> $APLSIZE 1);
BODY:
   EXECUTE print "In metalevel plan!  APL is:\n";
   EXECUTE printAPL $APL;

   // Find lowest-cost element
   ASSIGN $COUNT 1;
   ASSIGN $LOWESTINDEX -1;
   ASSIGN $LOWESTCOST 999999.0;
   WHILE : TEST (<= $COUNT $APLSIZE)
   {
         EXECUTE getAPLElement $APL $COUNT
               $APLELEMENT;
         EXECUTE getAttributeValue $APLELEMENT "Cost"
               $VALUE;
         WHEN : TEST (< $VALUE $LOWESTCOST)
         {
               EXECUTE print "Found new lowest cost APL
                     Element at #" $COUNT "\n";
               ASSIGN $LOWESTINDEX $COUNT;
               ASSIGN $LOWESTCOST $VALUE;
         };
         ASSIGN $COUNT (+ $COUNT 1);
   };

   OR
   {
         TEST (!= $LOWESTINDEX -1);
         EXECUTE print "Lowest cost APL Element is #"
               $LOWESTINDEX "\n";
         EXECUTE getAPLElement $APL $LOWESTINDEX
               $APLELEMENT;
   }
   {
   // If no lowest-cost element then pick randomly
   EXECUTE print "No lowest-cost element, picking
         randomly.\n";
   EXECUTE selectRandomAPLElement $APL $APLELEMENT;
   };

   EXECUTE print "Intending APL Element:\n";
   EXECUTE printAPLElement $APLELEMENT;
   EXECUTE intendAPLElement $APLELEMENT;

EFFECTS:
   EXECUTE print "In metalevel plan! Retracting WM entry for
this level.\n";
   RETRACT APL $LEVEL;
   EXECUTE print "In metalevel plan! Retracting WM entry for
previous level.\n";
   RETRACT APL (- $LEVEL 1);

FAILURE:
   EXECUTE print "\n\nMetalevel plan failed!\n\n";
}
```

Figure 5. A data-driven plan for performing metalevel reasoning.

Agent programmers can augment the functionality provided with JAM by defining primitive functions in native Java code and several access methods to the Java code are provided by the JAM architecture. It is through this augmentation of primitive functions that provide JAM with application-specific (e.g., database interfacing) and "social" abilities (such as interagent communication and collaborative). There are a number of predefined primitive actions included with the JAM agent distribution, including those providing debugging support and agent mobility (which we describe in more detail below).

2.4 WORLD MODEL

The JAM World Model holds the facts that represent the current state of the world as it is known by the agent. Information that might be kept there includes state variables, sensory information, conclusions from deduction or inferencing, modeling information about other agents, etc. Each world model entry is a simple proposition of the form:

relation_name argument1 argument2 ... argumentN;

The ordering, semantics, and typing of the arguments is unconstrained and is determined by the agent programmer. World model relation's arguments are currently limited to the following types: strings, floating point numbers, integer numbers, and native Java objects. Specification of an agent's initial world model consists of creating a text file containing the keyword "FACTS:" followed by the list of world model relations. The agent parses the initial world model specification before execution begins and can make assertions, retractions, and modification dynamically within plans. We show an example of an agent's initial World Model below in Figure 6.

```
FACTS:
      FACT ON "Block5" "Block4";
      FACT ON "Block4" "Block3";
      FACT ON "Block1" "Block2";
      FACT ON "Block2" "Table";
      FACT ON "Block3" "Table";
      FACT CLEAR "Block1";
      FACT CLEAR "Block5";
      FACT CLEAR "Table";
      FACT initialized "False";
```

Figure 6. Example World Model for a blocks-world domain.

2.5 OBSERVER

The *observer* is an optional declarative procedure that the JAM interpreter executes between each action in a plan. The observer procedure itself is basically a plan with only the plan body and none of the other plan components and is specified in the form:

> *OBSERVER: {*
> *[non-subgoaling procedure]*
> *}*

The observer represents an architectural hook which an agent programmer can use to implement capabilities that are more easily implemented outside of the scope of JAM's normal goal/plan-based reasoning. Examples of such capabilities might be to check a buffer of incoming messages to see if any new messages have arrived or to see if the agent's external vision processing component has buffered new images. We named this procedure "observer" because of its typical use in watching for asynchronous events. Because the observer procedure is executed very frequently, it should not be computationally intensive.

We show an example of an agent's Observer procedure below, in Figure 7. The procedure performs an initialization function upon startup and periodically checks for incoming messages from other agents and asserts any messages to the agents world model when received.

```
OBSERVER:
{
  RETRIEVE cycle_num $CYCLE_NUM;
  UPDATE (cycle_num) (cycle_num (+ 1 $CYCLE_NUM));
  RETRIEVE lastTime $LASTTIME;
  EXECUTE getTime $CURRENTTIME;

  WHEN : TEST (== $CYCLE_NUM 0) {
    EXECUTE perform_initialization;
  };

  WHEN : TEST (> (- $CURRENTTIME $LASTTIME) 5000) {
    EXECUTE getMessages $MSGS $NUM_MSGS;
    ASSIGN $MSG_NUM 1;
    WHILE : TEST (<= $MSG_NUM $NUM_MSGS) {
      EXECUTE getMessage $MSGS $MSG_NUM $MSG;
      ASSERT new_message $MSG;
      ASSIGN $MSG_NUM (+ 1 $MSG_NUM);
    };
    UPDATE (lastTime) (lastTime $CURRENTTIME);
  };
}
```

Figure 7. Example Observer procedure demonstrating using it for initialization and periodic checking for messages

Note that all of the functionality and behavior embedded into an observer procedure can be implemented in JAM's normal BDI paradigm (using goals and plans) to take advantage of the architectures powerful reasoning capabilities. For example, goals and plans can be written to determine the best time and manner in which to check for communication queues for new messages. It may not always be pragmatic to implement all activities using the BDI paradigm however and we have found that the observer paradigm is very useful.

3. JAM AGENT CHECKPOINTING and MOBILITY

JAM agents facilitate building applications requiring mobility through the use of *checkpointing* capabilities. That is, we have implemented functionality for capturing the runtime state of a JAM agent in the middle of execution and functionality for subsequently restoring that captured state to its execution state. One use of this functionality is for periodically saving the agent's state so that it can be restored in case the agent fails unexpectedly. This facilitates building robust applications that can restart and recover from otherwise catastrophic termination. Another use of the checkpointing functionality is to implement agent mobility, where the agent creates a checkpoint and restores it to execution on a different computer. A third possible use of this functionality is to clone an agent by creating a checkpoint and restoring it to an execution state without terminating the original agent. In all cases, a simple Java class is provided with JAM that performs the basic restoration function. Extension of this restoration class to provide application-specific mobility policies and similar functionality can be made as needed.

We have simplified agent mobility by implementing an *agentGo* primitive function. This function allows an agent programmer to simply specify a target computer and port and, when the plan containing the function is executed, the agent will transfer to the other machine and terminate itself on the initial computer. On the destination machine, the JAM agent will resume execution, guided by its pre-existing goals and plans, from where it was suspended on the initial machine. Moving between computers therefore becomes transparent in the sense that such activity is not handled differently than any other activity. The example plan shown in Section 2.3 illustrates a plan using the agentGo primitive.

4. CONCLUSIONS AND FUTURE WORK

We believe JAM represents the current leading edge in pragmatic BDI-theoretic intelligent agent architectures. JAM provides rich, expressive procedural representations, a wide range of useful goal semantics, metalevel reasoning, support for complex utility-theoretic behavior, and agent mobility support while remaining true to its underlying BDI theoretics.

JAM does not represent a complete architecture yet, however, in that many architecturally integrated capabilities such as plan generation and learning do not yet exist. Towards the end of a JAM architecture that includes these capabilities, we are currently in the middle of adding generative planning functionality to the JAM interpreter, so that when JAM reaches an impasse (in Soar terminology), it can generate a plan from first principles (using a novel hybrid HTN and partial order planning algorithm) rather than relying solely upon the library of pre-programmed plans as most BDI architectures (e.g., PRS-CL and UMPRS). We have extended the JAM plan representation to include declarative representations for individual primitive actions and are implementing partial order planning algorithms based upon the new representations. "Social" abilities in the form of conversation management and FIPA-compliant language and protocol support have been added to JAM as an application-specific extension, but we have not yet decided upon whether such capabilities will become an integral part of JAM at any point or if such functionality will be provided as a supplemental package.

5. ACKNOWLEDGEMENTS

We would like to thank Jaeho Lee for his help in implementing portions of the original version of JAM and for many discussions regarding PRS and BDI architectures. We would also like to acknowledge the Johns Hopkins University's Applied Physics Laboratory which supported some of the development of JAM.

6. REFERENCES

[1] P. R. Cohen, and H. J. Levesque. Intention = Choice + Commitment. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, Seattle, Washington, 410-415, 1987.

[2] E. H. Durfee, M. J. Huber, M. Kurnow, and J. Lee. TAIPE: Tactical Assistants for Interaction Planning and Execution. In *Proceedings of the First International Conference on Autonomous Agents*, 443-450, Marina del Rey, CA, 1997.

[3] I. A. Ferguson. *TouringMachines: An Architecture for Dynamic, Rational Mobile Agents*. Ph.D. Thesis, University of Cambridge, UK, 1992.

[4] R. E. Fikes and N. J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. In *Artificial Intelligence Journal*, Vol 2, 189-208, 1971.

[5] E. Gat. Integrating Planning and Acting in a Heterogeneous Asynchronous Architecture for Controlling Real-World Mobile Robots. In Proceedings of the Tenth National Conference on Artificial Intelligence, pages 809-817, San Jose, CA, 1992.

[6] M. Georgeff and A. L. Lansky. Reactive Reasoning and Planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, 677-682, Seattle, Washington, 1987.

[7] R. S. Gray, D. Kotz, G. Cybenko, and D. Rus. Agent Tcl. In W. Cockayne and M. Zyda editors, *Mobile Agents*, Manning Publishing, 1997.

[8] M. J. Huber and T. Hadley. Multiple Roles, Multiple Teams, Dynamic Environment: Autonomous Netrek Agents. In *Proceedings of the First International Conference on Autonomous Agents*, pages 332-339, Marina del Rey, CA, 1997.

[9] M. J. Huber, J. Lee, P. Kenny, and E. H. Durfee, UM-PRS Programmer and User Guide, The University of Michigan, Ann Arbor MI 48109, 1993. [See `http://members.home.net/marcush/IRS`].

[10] F. Ingrand, M. Georgeff, and A. Rao. An Architecture for Real-Time Reasoning and System Control. *IEEE Expert*, 7(6):34-44, 1992.

[11] P. G. Kenny, E. H. Durfee, and K. C. Kluge. Mission Planning and Coordinated Execution for Unmanned Vehicles. In *Proceedings of the Sixth Computer Generated Forces and Behavioral Representation Conference*, 329-335, 1996.

[12] J. E. Laird, A. Newell, and P. S. Rosenbloom. SOAR: An Architecture for General Intelligence. *AI Journal*, 1-64, 1987.

[13] J. Lee, M. J. Huber, E. H. Durfee, and P. G. Kenny. UM-PRS: An Implementation of the Procedural Reasoning System for Multirobot Applications. In *Conference on Intelligent Robotics in Field, Factory, Service, and Space (CIRFFSS'94),* 842-849, Houston, Texas, 1994.

[14] J. Lee and E. H. Durfee. Structured Circuit Semantics for Reactive Plan Execution Systems. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, 1232-1237, 1994.

[15] J. P. Muller and M. Pischel. Integrating Agent Interaction into a Planner-Reactor Achitecture. In *Proceedings of the 1994 Distributed AI Workshop*, pages 250-264, Lake Quinalt, WA, 1994.

[16] K. L. Myers and D. E. Wilkins. The Act Formalism, Version 2.2. SRI International Artificial Intelligence Center Technical Report, Menlo Park, CA, 1997.

[17] K. L. Myers, User Guide for the Procedural Reasoning System, SRI International AI Center Technical Report, SRI International, Menlo Park, CA, 1997.

[18] A. Newell. *Unified Theories of Cognition*, Harvard University Press, 1990.

[19] A. S. Rao and M. P. Georgeff. Modeling Rational Agents Within a BDI-architecture. In *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*, Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[20] A. S. Rao and M. P. Georgeff. A Model-Theoretic Approach to the Verification of Situated Reasoning Systems. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, 318-324, Chambery, France, 1993.

[21] Y. Shoham. Agent-oriented Programming. *Artificial Intelligence*, 60(1):51-92. 1993.

[22] Y. Shoham. AGENT0: A Simple Agent Language and Its Interpreter. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, 704-709, Anaheim, California, 1991.

[23] R. S. Thomas. The PLACA Agent Programming Language. In *Intelligent Agents - Theories, Architectures, and Languages*, pages 356-370, Michael Wooldridge and Nicholas R. Jennings editors, Springer-Verlag, 1995.

[24] J. M. Vidal and E. H. Durfee. Task Planning Agents in the UMDL. In *Proceedings of the 1995 CIKM Intelligent Information Agents Workshop*, 1995.

[25] D. E. Wilkins and K. L. Myers. A Common Knowledge Representation for Plan Generation and Reactive Execution. In *Journal of Logic and Computation*, vol. 5, number 6, 731-761, 1995.