Bringing Planning to Autonomic Applications with ABLE

Biplav Srivastava
IBM India Research Laboratory
Block 1, IIT Delhi, Hauz Khas,
New Delhi 110016, India.
Email: sbiplav@in.ibm.com

Joseph P. Bigus and Donald A. Schlosnagle
IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights,
New York, 10598, USA.
Email: {bigus, daschlos}@us.ibm.com

Abstract

Planning has received tremendous interest as a research area within AI over the last three decades but it has not been applied commercially as widely as its other AI counterparts like learning or data mining. The reasons are many: the utility of planning in business applications was unclear, the planners used to work best in small domains and there was no general purpose planning and execution infrastructure widely available. Much has changed lately. Compelling applications have emerged, e.g., computing systems have become so complex that the IT industry recognizes the necessity of deliberative methods to make these systems self-configuring, self-healing, self-optimizing and self-protecting. Planning has seen an upsurge in the last decade with new planners that are orders of magnitude faster than before and are able to scale this performance to complex domains, e.g., those with metric and temporal constraints. However, planning and execution infrastructure is still tightly tied to a specific application which can have its own idiosyncrasies. In this paper, we fill the infrastructural gap by providing a domain independent planning and execution environment *that is implemented in the ABLE agent building toolkit, and demonstrate its ability to solve practical business applications. The planning-enabled ABLE is publicly available and is being used to solve a variety of planning applications in IBM including the self-management/autonomic computing scenarios.*

1 Introduction

Planning has received tremendous interest as a research area within AI over the last three decades. However, compared to its other AI counterparts like learning or data mining, it has not been applied widely in commercial applications. Earlier, the utility of planning in business applications was unclear. Most success stories were in specialized domains like NASA's RAX planner[6] for space shuttle mission control and SIPE in production line scheduling of a beer factory[13, 17]. Today, more and more applications of planning are emerging in areas as diverse as pathway discovery in bioinformatics[10], data integration[11] and web services composition in application middleware[15, 2], organizing marketing campaigns[18], to elevator control[9] and autonomic computing[1]. The latter is IBM's vision for self-managed complex systems which has been accepted by much of the IT industry and calls for the systems to be self-configuring, self-healing, self-optimizing and self-protecting. In terms of performance, planning has seen an upsurge in the last 6-7 years with new planners that are orders of magnitude faster than before and are able to scale this performance to complex domains, e.g., metric and temporal constraints.

The success of planning in an application depends as much on the planning techniques used as on the way it is embedded into the runtime. The former represents a knowledge engineering challenge since the formulation of the planning problem (i.e., goal and initial states, actions, plan evaluation metric) should be meaningful in the domain assuming that an appropriate planning technique would be used. The latter refers to development of tools and methodologies to integrate off-line reasoning with runtime so that the most important and useful planning scenarios get efficiently solved. This is the focus of the paper.

Planning and execution is still tightly tied to the specific context of the individual application which can have its own idiosyncracies. This lack of domain independent planning and execution infrastructure makes it hard to understand the role of planning in an application (are the planning needs really special or was the implementation ad hoc?), slows future upgradation to planning advancements and inhibits solution reuse. Business applications today, however, are built along architectures that allow componentization of building blocks, large-scale reuse and easy upgradation/maintenance. Hence, a domain independent plan-

[1] http://www.research.ibm.com/autonomic/

ning and execution framework is needed for widely applying planning.

As an example of business application, we will consider the problem of Automated System Recovery of web applications that are running behind a website. In this scenario, the web applications should be able to automatically self-configure and self-heal in response to runtime exigencies to keep the website available. As a very simple problem instance, let the website run on two high-end machines. The website uses two applications (e.g., servlets) that can run on any of the two machines provided an application server (e.g., WebSphere Application Server) is running on that machine. The applications access a database server (e.g., DB2) and a directory server (e.g., SecureWay), which may run on any machine. The initial state has the two machines running and all the software servers installed. The goal is to have both the applications running over time. If any software server or machine were to fail, the system should be able to infer and initiate actions to migrate the computation in a way that the web applications continue to run. A typical business website runs tens of applications on similarly large number of machines and follows a multi-tiered architecture integrating components from differents part of the firm's business - Products, Accounting and Finance, Supply Chain, Customer Relationship Management, etc.

In this paper, we fill the infrastructural gap by providing a domain independent *planning and execution environment* that is implemented in the IBM Agent Building and Learning Environment (ABLE) toolkit[1]. ABLE[2] is a toolkit for building multiagent autonomic systems. It provides a lightweight Java agent framework, a comprehensive JavaBeans library of intelligent software components, a set of rule development and test tools, and an agent platform. ABLE supports various type of rules (e.g., If-then-else, Fuzzy rules, Prolog rules) and their corresponding rule engines. A developer can build a composite JavaBean (called an *agent*) by mixing different types of rules and embed the resulting component in an application.

We have extended ABLE with a new type of rule that we call planning rules, and it is compliant with the planning community's Planning Domain Description Language (PDDL[5]). Since PDDL comes in various flavors, i.e. levels, the planning rules cannot be tied for processing to any specific planning engine. Therefore, we have developed a general planning framework called Planner4J [14] comprising of a set of common interfaces[3], and any planner that is compliant with it can be used to process the planning rules, provided it can handle the corresponding level of expressivity (e.g., PDDL level). The planning framework also provides a common infrastructure so that a variety of planners are available to the developer and new ones can be easily built by reusing much of the existing code. As proof of concept, we have implemented a PDDL1 classical planner and a limited metric planner.[4]. The key benefits of using the planning-enabled ABLE are:

- It provides the applications with a common planning and execution platform to embed, test and evolve with state-of-the art planners.
- It supports arbitrary customization of an action's execution-time behavior using Java methods. Furthermore, the action set can be modified in the dynamic environment and a new planning problem posed quite easily.
- It contains a planning framework to develop new planners by reusing existing components.
- The existing range of learning beans, rule types and data filters can be used to build complex planning agents.

To our knowledge, ABLE is the first publicly available toolkit that provides general purpose planning and execution support. It is being used to solve a variety of planning applications in IBM including the self-management/autonomic computing scenarios.

Here is the outline of the paper: we start with a brief description of ABLE, business policies, and characterize planning as goal-based business policies that can be realized with ABLE rules. We then describe how the planning problems can be input to ABLE in both PDDL and ABLE Rule Language (ARL) and describe the planning process. We then give a summary of Planner4J framework and demonstrate planning and execution in the system recovery example using ABLE. Next, we discuss related work and argue how the current system can easily bring planning to commercial applications. Finally, we conclude with pointers to future work.

2 Background

At the outset, we clarify the meaning of two common terms - (business) policies and rules - used in the paper. Policy is a popular term in industry refering to any declarative specification of behavior that is desired from a software system (e.g., agent). We use *rule* to mean the declarative representation used to realize policies and ABLE Rule Language (ARL) is its example.

[2] Available at http://www.alphaworks.ibm.com/tech/able.

[3] It also has utility functions and reference planner implementations.

[4] The ABLE 2.0.1 version on Alphaworks contains only the classical planner.

2.1 Policies and Planning

The behavior of a system can be specified by two types of policies. In the first case, a policy can exhaustively list conditions and specify the corresponding actions that need to be taken. During runtime, a policy engine will verify the conditions and take the stipulated action. This type of policy is procedural in nature and suited for reactive reasoning. In the second case, the policy only lists the system's expected behavior (e.g., goal state) and it is left to the policy engine to deliberate and determine what actions need to be taken to ensure the satisfaction of goals. A generalization of goal type policy can include utility information so that the selection of actions depends on runtime situations.

Planning can be seen as the technology for enforcing goal type policies. A planning problem P is a 3-tuple $\langle I, G, A \rangle$ where I is the complete description of the initial state, G is the partial description of the goal state, and A is the set of executable (primitive) actions. A planner will find a possible solution to P, which is an action sequence S, such that if S is executed in I, the resulting state of the world would contain G.

Planning is a very wide discipline characterized by how the environment, the agent's goal and its model of the world are represented. Planning algorithms are best understood as a refinement search over sets of possible plans - an algorithm starts from the set of all possible plans and performs refinements on the plan set leading to sub-sets from which extracting a single solution is feasible[8, 7]. Various planners can return sequential, parallel or optimal plans with respect to a defined metric.

2.2 ABLE

The ABLE Rule Language (ARL) is a rule-based programming language that provides tight integration with Java objects and the ability to externalize business logic using simple business rules or more complex inferencing rules. ABLE provides a set of rule engines ranging from lightweight procedural scripting to medium weight forward and backward chaining, up to the power and speed of advanced AI inferencing techniques. ARL also supports templates to allow customization of rulesets by non-technical users.

Rules: A rule is a declarative statement or knowledge expression. In ARL, all statements are referred to as rules, which can be roughly divided into two types, scripting and inference rules. Scripting rules include assertion (assignment) rules, if-then-else rules, for-loop rules, while-do, do-while, and do-until iteration rules. Inference rules include if-then rules, when-do pattern match rules, predicate logic and now, planning rules.

Rule Blocks: Multiple rules can be grouped together into a rule block. Each rule block has an associated inference engine that interprets the rules in that block. This allows the user to mix multiple inference techniques (for example, forward and backward chaining) with procedural scripts. The inference engines implement the control strategies that affect how the rules are interpreted.

```
ruleset PddlFilePlanTest {
predicates{} ;
variables {
    // One variable minimum
    String message = new String("Invoking planner!");
};
inputs{};
outputs{};
void process() using Script {
     Rule0: println(message); // One rule minimum
          : invokeRuleBlock("runplanner");
}
void runplanner() using Planning {
// solve problem based on the pddl files
     : setControlParameter(ARL.DomainFile,
                           ≺ domain.pddl ≻);
     : setControlParameter(ARL.ProblemFile,
                           ≺ problem.pddl ≻);
}
}
```

Figure 1. *RuleSet to invoke planning problems in PDDL using ABLE. Of interest are the "process" and "runplanner" ruleblocks that are processed by Script and Planning rule engines, respectively.*

RuleSets: An ABLE Rule Language ruleset contains data, ruleblocks and rule specifications. They can take the form of text files with Java-like syntax, of XML documents conforming to the ABLE ruleset XML schema definition, or of serialized AbleRuleSet JavaBeans. An example of an ABLE ruleset is shown in Figure 1.

Inference Engines: An inference engine is a control algorithm that processes a group of rules (in a ruleblock). Each inference engine implements an algorithm for deriving a result given a set of input data and the corresponding rules in the rule block. ABLE provides a set of rule engines that can be used to process ARL rulesets and also can be extended via pluggable user-defined rule engines. The ABLE rule engines include:

1. **Script** (procedural): sequential evaluation of rules
2. **Backward chaining**: goal-chaining through if-then rules
3. **Predicate backward chaining**: goal-chaining with backtracking using predicate rules

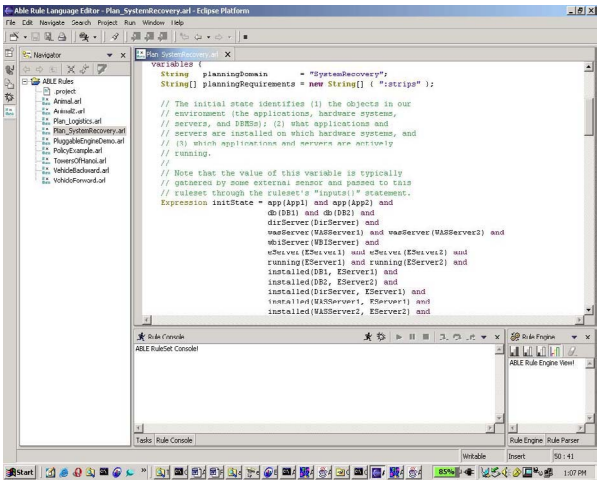

Figure 2. *Building a planning rule (planning domain and problem) with ABLE.*

4. **Fuzzy forward chaining**: multi-step chaining over fuzzy if-then rules with linguistic variables and hedges

5. **Forward chaining**: data-driven inferencing over if-then rules

6. **PatternMatch**: forward chaining with pattern matching - data-driven inferencing over when/do rules. Also contains **PatternMatchRete** which uses a Rete' network for implementation.

7. **Planning**: uses planning techniques to find a sequence of actions to move from an initial state to a goal state. A planning framework (Planner4J) is in place to integrate classical planning and more complex algorithms.

In ABLE, rule engines are coded in Java and compiled into bytecodes. Rules are compiled into Java objects and are processed by the inferencing engine specified for a rule block. At execution time, ABLE can execute any action recommended by the inference engine(s) by invoking appropriate Java objects associated with the rules.

3 Planning Rules

Planning problems can be input to ABLE using the ARL planning rules. ABLE provides a rule authoring environment (plugin) in Eclipse[5] IDE for developing and testing rules. Figure 2 shows the planning rule being developed for the system recovery example.

If a user already has the planning domain and problem information in PDDL, the planner can be invoked directly with minimal effort. The ruleset shown in Figure 1 specifies the domain and problem file for the planner to use (through control parameters) and spew out the output. This is a fast way for someone to start using the planner within ABLE without learning ARL.

However, the full power of ABLE can be realized by coding the planning problem in ARL since it allows the user to mix rules of different types and invoke arbitrary Java objects for realizing actions. The ARL for an example system recovery problem is shown in the Appendix.

The ARL consists of specifications about predicates, variables, expressions, inputs and outputs. The important rule blocks are `init()` to initialize the domain information, `process()` to initialize problem information and invoke the planner, and `postProcess()` to cleanup. Control parameters are provided to specify the Objects in the planning problem, and the initial and goal states. In the planning rule block (called `doPlanning()` in the example), each action in the domain is specified as a rule. The planning rule semantics are equivalent to the Planning Domain Definition Language (PDDL) action and very similar in syntax.

The Planning engine(s) incorporated in ABLE process Assertion and Planning rules. This is elaborated more in the next section. The processing sequence is to:

1. Process all Assertion rules in their declaration order.

2. Build a planning problem specified by the initial condition predicates.

3. Select and fire planning rules (actions) until the goal state is reached.

4. Return a plan solution, a sequence of actions with parameters to operationalize the plan.

4 Planner4J Planning Framework

Planner4J is an architectural framework for building planners[14]. It consists of important interfaces that are required to define and solve a planning problem, viz., action, state, problem, domain and plan, and their reference implementations that can be further reused and extended. The intention on the usability front is to provide the user (planning application/rule developer) with a consistent view of planners so that they are not restricted to any one type of algorithm and can easily select the best-in-class, efficient and expressive planner that they may need over time. On the software engineering front, the aim is to build a common infrastructure so that while developing new planners, much of the existing components (existing code) can be reused.

The Planner4J architecture is arranged as a layering of modules. Planner4J-Core is the core module made up

[5] http://www.eclipse.org

Package	Interfaces	Implementations
≺prefix≻.planners	IPlanner IPredicate IDomain IProblem IAction IPlanSolution	Options
≺prefix≻.parsers	IParser	
≺prefix≻.state	IState	
≺prefix≻.search	ISearch	SearchQueue

Table 1. Planner4J-Core Components.

Package	Implementations	Helpers
≺prefix≻. planners	ClassicalPlanner PredicateImpl DomainImpl ProblemImpl ActionImpl PlanSolutionImpl	ActionManager PredicateManager HelperUtil
≺≻.parsers	PDDL1Parser	
≺≻.state	StripsStateImpl	
≺≻.search	StateSpaceSearchImpl	
≺≻.test	TestClassicalPlanner	

Table 2. Planner4J-Classical Components.

of packages containing generic interfaces and implementations of common capabilities like command-line processing and search queue(see Table 1). All Planner4J planners share the Planner4J-Core, thus promoting reuse and extensibility in the Planner4J design.

The Planner4J-Classical module contains the reference implementation for the Planner4J-Core interfaces (see Table 2). It implements a "classical planner" (STRIPS) that can solve planning problems represented in PDDL1. The module also contains an additional package to house test case driver programs. Planner4J allows additional planners to be built using the Planner4J-Core and Planner4J-Classical modules.[6]

Each package contains implementations of interfaces defined in the corresponding package of Planner4J-Core module. In the planner package, ClassicalPlanner implements the IPlanner interface and drives the planning process. Additionally, helper classes are defined to manage action and predicate objects. The StripsStateImpl implementation of IState contains routines to record information about literals and reason with states. The StateSpaceSearchImpl class implements a forward as well as a backward state space search regime guided by a heuristic function. The direction of the search is customizable with a switch provided in the Planner4J-Core Options class. Each planner implementation contains a test program to illustrate how the planner can be programmatically invoked.

Package	Implementations	Helpers
≺prefix≻. planners	MetricPlanner MetricActionImpl MetricPlanSolutionImpl	HelperUtil
≺≻.parsers	MetricParser	
≺≻.search	MetricStateSpaceSearchImpl	
≺≻.test	TestMetricPlanner	

Table 3. Planner4J-Metric Components.

At the heart of the implemented planners is a heuristics-driven state space search algorithm which can be tuned to forward or backward search direction. Different types of planning (Classical, Metric or HTN[7]) change the representation of the actions and the states. This effect can be modeled as changing the heuristic calculation function that is used to measure distance between states of the underlying planner.

Now consider how a new planner may be developed. Also implemented in Planner4J is a restricted metric temporal planner that can reason about cost and performance of actions in generating a feasible plan. It reuses the implementation of Planner4J-Classical as much as possible. In Table 3, the components of the Planner4J-Metric are summarized.

MetricPlanner implements the IPlanner interface. In comparison to PDDL1, actions in this case contain annotations compliant with PDDL2 for cost and duration of actions. As a result, the file parsing routines (MetricParser), the action implementation (MetricActionImpl), the implementation for planning solution (MetricPlanSolutionImpl), and the heuristic evaluation function (MetricStateSpaceSearchImpl) have to be primarily extended/changed. However, even these implementations can reuse the corresponding implementations in Planner4J-Classical.

In the general representation of metric temporal planning, predicates can have duration. We will support this in future and that would require extensions to predicate and state related implementations.

The reader should note that though the existing planners are based on heuristic search space approach, and can of course be improved with known better optimizations/heuristics (or novel ones), *the Planner4J architecture itself is a set of interfaces which does not dictate any particular implementation approach*. We will encourage external planner contributions to extend the spectrum of readily-

[6]Note that the use of Planner4J-Classical module is optional but the interfaces in Planner4J-Core must be implemented.

[7]Not an exclusive list. We have identified other types that we want to implement in future.

available planners. One restriction we do impose is that the planner be implemented in Java.

5 Demonstration

The ABLE distribution contains a working example of how a plan can be processed. In the discussion in this section, we are refering to Plan_SystemRecovery.arl and SystemRecoveryActions.java in the com/ibm/able/examples/rules directory of the ABLE installation and also listed in the appendix.

Since actions to carry out a plan are always domain specific, one must provide these actions as methods in a Java class that you write. The methods must be public, static, return a boolean indicating whether the action worked (true) or not (false), and correspond one-for-one to the planning rules in the ARL file. For example, given the ARL planning rule on the left below, the signature of the method in the Java domain actions class must be as shown on the right:

Plan_SystemRecovery.arl

```
startDB:
  parameters
    (Object DB,
    Object EServer)
  precondition
    (db(DB) and
    not(running(DB)) and
    running(EServer) and
    installed(DB, EServer))
  effect
    running(DB);
```

SystemRecoveryActions.java

```
public static boolean
  startDB(String theDB,
    String theEServer)
  { ... }
```

There is an important thing to note about the Java class: even though the ARL parameter list can specify any data type, the classical planning engine currently only provides Strings on plan output; so all Java method parameters must be of type String. This is reflective of Planner4J's origin from conventional planning where the information about an object's type is just a label and only used to instantiate and verify predicates and actions. But Java allows many sophisticated type-based operations, and supporting abstract data types will be an area of future work. Once one has written the Java class that knows how to perform the planning actions unique to the problem domain, one must import that class into the planning rule (ARL) file:

```
import com.ibm.able.examples.rules.SystemRecoveryActions;
```

We also tell the planning inference engine that a Java class is available for processing the generated plan:

```
:setControlParameter("doPlanning",ARL.Domain, planningDomain);
:setControlParameter("doPlanning",ARL.Requirements,
                     planningRequirements);
:setControlParameter("doPlanning",ARL.DomainActionsClass,
                     SystemRecoveryActions);
```

After a planning rule block has been used to create a plan, the plan can be processed by coding a few simple ARL statements. First, the generated plan is represented as a Java ArrayList, so one must import that data type into the ARL file:

```
import java.util.ArrayList;
```

Next, one needs to declare a variable into which the plan generated from the planning inference engine can be stored. We also declare a variable that is used to determine whether the plan was processed successfully:

```
// The output of the planning engine is retrieved to
// the following variable.
ArrayList thePlan;
// The result of processing the generated plan.
// True = success; false = failure.
Boolean returnCode;
```

Next, we obtain the plan and process it with the domain actions class:

```
// Obtain addressability to the generated plan
// and then process it. If the plan is empty,
// no actions will occur.
: thePlan = this.getControlParameter("doPlanning", ARL.Plan);
: returnCode = this.processPlan(thePlan);
: if (returnCode)
    then println("Plan processed successfully.");
else
    println("Plan processing encountered an error.");
```

Note that if any of the methods in the Java domain actions class return false, the plan processor will stop processing the plan and return control to the ruleset. If all actions return true, the entire plan will have been processed successfully. The planning actions can be as simple or as complicated as one wants to make them. If the plan fails, one may want to gather a new initial state, invoke the planning inference engine again, and then try to process the new plan.

The plan generated and executed for the system recovery problem is shown in Figure 3. Since the activity associated with each action in the Java actions file (see appendix) is only to print messages, this is what is shown in the screen shot.

6 Discussion

The planning area has seen a rush of applications recently. There is also a wide variety of planners available, e.g., LPG[8], Sapa[3], FF[9]. However, the success of planning in an application depends as much on the planning techniques used as on the way it is encoded and embedded into the runtime. Tools for building intelligent agents are few and those supporting domain-independent planning and execution are fewer. Soar[10] is a well known architecture in academia to build intelligent agents. It provides tools to build agents in multiple languages and provides support for knowledge representation and inferencing. ABLE is a Java-based toolkit customized for building autonomic and business applications incorporating a range of rules.

[8] http://zeus.ing.unibs.it/lpg/

[9] http://www.informatik.uni-freiburg.de/ hoffmann/ff.html

[10] http://www.eecs.umich.edu/ soar/

Figure 3. *The plan generated for system recovery problem instance and its subsequent execution.*

For planning, GIPO (Graphical Interface for Planning with Objects) is an experimental GUI and tools environment for building planning domain models[12]. It can serve a complementary role with ABLE of helping the user analyze the business environment so that a planning model can be built using the ARL representation.

There is a large body of work in policy-based systems, e.g., networking [16]. However, the policies are reactive in nature and ABLE allows these applications to support goal-based policies.

7 Conclusion and Future Directions

In this paper, we introduced a domain independent *planning and execution environment* that is suitable for building business applications which apply planning. The planning-enabled ABLE bridges the infrastructural gap between the new business applications that are emerging, and the need to treat planners with the same programmatic discipline for extensibility, revision, and reuse as any other software. To process the planning rules, a planning framework is in place consisting of core interfaces and a reference implementation of classical planner.

In the future, we intend to extend the planning and execution capabilities in two directions. On the planning front, we want to improve the current implemetation with better heuristics and tighter ABLE and Planner4J integration, provide newer types of planners (e.g., HTN[4]) and incorporate external planners e.g., Sapa. On the execution front, we want to include fine-grained plan monitoring and execution support so that partially-executed, world-altering actions can be taken into account during replanning.

8 Acknowledgements

We thank Jana Koehler, Richard Goodwin, Joe Hellerstein and Jeff Kephart for useful discussions on practical planning.

References

[1] Bigus, J., Schlosnagle, D., Pilgrim, J., Mills, W., and Diao, Y. 2002. ABLE: A Toolkit for Building Multiagent Autonomic Systems. *IBM Systems Journal, Volume 41, Number 3. Also at http://www.research.ibm.com/journal/sj/413/bigus.html.*

[2] Blythe, J., Deelman, E., Gil, Y., Kesselman, C., Agarwal, A., and Mehta, G. 2003. The Role of Planning in Grid Computing. *Proc. Intl Conf. on Automated Planning and Scheduling (ICAPS).*

[3] Do, B., and Kambhampati, S. 2001. Sapa: A Domain-Independent Heuristic Metric Temporal Planner. *Proc. European Conference on Planning.*

[4] Erol, K. 1995. Hierarchical task network planning: Formalization, Analysis, and Implementation. *Ph.D. thesis, Dept. of Computer Science, Univ. of Maryland, College Park, USA.*

[5] Fox, M., and Long, D. 2002. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Available at http://www.dur.ac.uk/d.p.long/competition.html.*

[6] Jonsson, A. K., Morris, P. H., Muscettola, N., Rajan, K., and Smith, B. D. 2000. Planning in Interplanetary Space: Theory and Practice. *Proc. Artificial Intelligence Planning Systems, Pg. 177-186, Also at citeseer.nj.nec.com/jonsson00planning.html.*

[7] Kambhampati, S., Knoblock, C. and Yang, Q. 1995. Planning as Refinement Search: A Unifying framework for evaluating design trafeoffs in partial order planning. *Artificial Intelligence, Special issue on Planning and Schedling. Vol. 76.*

[8] Kambhampati, S., and Srivastava, B. 1995. Universal Classical Planner: An algorithm for unifying state space and plan space approaches. *In New Trend in AI Planning: EWSP 95, IOS Press.*

[9] Koehler, J. and Schuster, K. 2000. Elevator Control as a Planning Problem. *Proc. AIPS 2000, pages 331-338.*

[10] Khan, S., Gillis, W., Schmidt, C. and Decker, K. 2003. A Multi-Agent System-driven AI Planning Approach to Biological Pathway Discovery. *Proc. Intl Conf. on Automated Planning and Scheduling (ICAPS).*

Proceedings of the International Conference on Autonomic Computing (ICAC'04)
0-7695-2114-2/04 $20.00 © 2004 **IEEE**

[11] Knoblock, C., Minton, S., Ambite, J., Ashish, N., uslea, I., Philpot, P., and Tejada, S. 2001. The Ariadne Approach to Web-based Information Integration. *International Journal on Cooperative Information Systems (IJCIS) 10 (1-2) Special Issue on Intelligent Information Agents: Theory and Applications, pp 145-169, 2001.*

[12] McCluskey, T.L., Liu,L., and Simpson, R. 2003. GIPO II: HTN Planning in a Tool-supported Knowledge Engineering Environment. *Proc. Intl Conf. on Automated Planning and Scheduling (ICAPS).*

[13] Production Line Scheduling in SIPE-2. 1995. *http://www.ai.sri.com/ sipe/beer.html.*

[14] Srivastava, B. 2004. A Software Framework for Applying Planning Techniques. *IBM Technical Report RI04001.* Available at http://domino.watson.ibm.com/library/CyberDig.nsf/Home.

[15] Srivastava, B. and Koehler, J. 2003. Web Service Composition: Current Solutions and Open Problems. *ICAPS 2003 Workshop on Planning for Web Services, pages 28 - 35.*

[16] Verma, D. C. 2001. Policy-Based Networking: Architecture and Algorithms. *New Riders Publ., ISBN: 1-57870-226-7.*

[17] Wilkins, D. E. 1990. Can AI planners solve practical problems? *Computational Intelligence, Vol. 6, No. 4, Pg. 232–246.*

[18] Yang, Q. and Cheng, H. 2003. Planning for Marketing Campaigns. *Proc. Intl Conf. on Automated Planning and Scheduling (ICAPS).*

A ARL for System Recovery Example (Plan_SystemRecovery.arl)

```
/**
 * This ABLE ruleset uses the planning engine to recover
 * from failures in an enterprise application infrastructure.
 */
ruleset Plan_SystemRecovery {

  predicates {
     installed, running, stopped, app, db, was_server,
     dir_server, eServer, wbi_server
  }

  variables {
    String   planningDomain       = "SystemRecovery";
    String[] planningRequirements = new String[] { ":strips" };

//  Objects app1 app2 db1 db2 eServer1 eServer2 dir_server
//  was_Server1 was_Server2 wbi_Server
    Expression initState_1 =
         app(App1) and app(App2) and db(DB1) and db(DB2) and
         dir_server(Dir_Server) and was_server(WAS_Server1) and
         was_server(WAS_Server2) and wbi_server(WBI_Server) and
         eServer(EServer1) and  eServer(EServer2) and running(EServer1) and
         running(EServer2) and installed(DB1, EServer1) and
         installed(DB2, EServer2) and installed(Dir_Server, EServer1) and
         installed(WAS_Server1, EServer1) and
         installed(WAS_Server2, EServer2) and
         installed(WBI_Server, EServer1) and installed(App1, EServer1) and
         installed(App2, EServer2) and running(DB1) and
         running(Dir_Server) and running(WBI_Server);

    Expression goalState_1 =
         running(App1) and running(App2);
  }

  inputs {}
  outputs{}

  /**
   * Set the planning Domain and Requirements.
   */
  void init() using Script {
    : setControlParameter("doPlanning", ARL.Domain, planningDomain);
    : setControlParameter("doPlanning", ARL.Requirements,
                          planningRequirements);
  }

  /**
   * Set the initial state, set the goal state,
   * and invoke the planning engine.
   */
  void process() using Script {
    // Problem: SystemRecovery-App2 , must also start
    // DB2 and WAS-Server2
    : setControlParameter("doPlanning", ARL.InitialState,
                          initState_1);
    : setControlParameter("doPlanning", ARL.GoalState,
                          goalState_1);
    : invokeRuleBlock("doPlanning");
  }

  /**
   * Specify rules for planning.
   */
  void doPlanning() using Planning {

   start_db:
      parameters ( Object DB,  Object EServer)
      precondition (
          db(DB) and not(running(DB)) and running(EServer) and
          installed(DB, EServer)
      )
      effect {
        running(DB);
      }

  start_was_server:
      parameters(Object WAS_Server, Object EServer, Object DB)
      precondition (
           was_server(WAS_Server) and not(running(WAS_Server)) and
           running(EServer) and installed(WAS_Server, EServer)and
           db(DB) and installed(DB, EServer) and running(DB)
      )
      effect {
        running(WAS_Server);
      }

  start_app:
      parameters(Object App, Object EServrver, Object DB,
                 Object Dir_Server, Object WAS_Server)
      precondition (
           app(App) and not(running(App)) and eServer(EServer) and
           running(EServer) and installed(App, EServer) and db(DB) and
           installed(DB, EServer) and running(DB) and
           dir_server(Dir_Server) and running(Dir_Server) and
           was_server(WAS_Server) and installed(WAS_Server, EServer) and
           running(WAS_Server)
      )
      effect {
        running(App);
      }
  }

  /**
   * Clean up.
   */
  void postProcess() using Script {
    : println("Done.");
  }
}
```

B An Action (*startwasserver*) in Java Action File (SystemRecoveryActions.java)

```
//====================================================================
// Imports
//====================================================================
/**
 * This class contains the planning actions specified in the ARL file
 * Plan_SystemRecovery.arl.
 */
public class SystemRecoveryActions implements Serializable {
  ...

  /**
   * Start the specified WAS server on the specified EServer.
   * @return true if the action is successful, false otherwise.
   */
  public static boolean startwasserver(String theWASServer,
       String theEServer, String theDB) {
    System.out.println("  \tSystemRecovery starting WAS Server<"+
          theWASServer+"> because...");
    System.out.println("  \t  \tEServer<"+theEServer+"> is running.");

    ...
    try {Thread.sleep(2000L);}
      catch(InterruptedException ex) { ; } // Do necessary action

    System.out.println("  \t  \tWAS Server<"+theWASServer+
          "> started successfully.");
    return true;
  }

...
}
```


Proceedings of the International Conference on Autonomic Computing (ICAC'04)
0-7695-2114-2/04 $20.00 © 2004 **IEEE**