The Case for Automated Planning in Autonomic Computing

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

Subbarao Kambhampati
Dept. of Computer Sc. & Engg.
Arizona State University
Tempe, AZ 85287, USA.
Email: rao@asu.edu

Abstract

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. Architectures for system self-management, also called Autonomic Computing (AC), have been proposed where elements are managed by monitoring and analyzing behaviors and using the response to plan and execute new actions that take or keep the system in desirable states. In this paper, we explore the planning needs of AC, its match with existing planning technology and its connections with policies and planning for web services and scientific workflows (grids). We show that planning is an evolutionary next step for AC systems that use procedural policies today. This connection also raises interesting research problems in adapting automated planning techniques to AC applications.

1 Introduction

The vision of Autonomic Computing (AC)[12] is to improve manageability of complex IT systems by making them self-configuring, self-healing, self-optimizing and self-protecting. This would require that the behavior of system elements are monitored and analyzed, and the performance is used to plan and execute suitable actions to take or keep the system in desirable states.

Policy is a popular term in industry to refer to any declarative specification of behavior that is desired from a software system (e.g., agent) and the behavior is enforced by a policy engine. Two types of policies are easily distinguishable. In the first case, the policy describes desired behavior and exhaustively lists necessary actions to meet them under all conditions. During runtime, a policy engine will verify the conditions and take the stipulated action. This type of policy is procedural in nature because the actions to take under a condition is fully known, and it is 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 provides the policy engines for goal-type policies.* Planning is thus critical for meeting the AC vision.

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[10, 9]. Various types of planners can return sequential[19], conditional plans or a generalized state-action mapping [5] specifying what action to take in any state during execution(hence procedural policies), that are optimized with respect to a defined metric. 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.

Despite the obvious potential for connections between automated planning and autonomic computing, very little has been done to exploit the synergy. Practitioners of AC computing tend to develop their own specialized and often brittle solutions for what are in essence planning problems, while researchers in the planning community seem to be unaware of the potential applications in autonomic computing.

The objective of this paper is thus two-fold:

- Motivate early adopters of AC, who are using procedural policies, to consider planning and goal-type policies by highlighting their potential for self-management.

- Foreground some of the challenges and research problems raised in adapting automated planning techniques to AC applications.

Here is the outline of the paper: we start with a brief overview of planning followed by description of AC scenarios and how planning can be useful there. Then we discuss two existing systems with planning capabilities demonstrated on AC scenarios - the first is an optimizing domain-dependent planning and scheduling system for self-configuration while the latter is a domain-independent planning and execution system. These case-studies are used to highlight the flexibility provided by automated planning technology, in contrast to procedural policies. We then identify AC-specific planning challenges. They include working with incomplete domain models and in managing life cycles of plans. We round up by relating planning to procedural policies and its connections to Web and Grid services and finally give closing comments.

2 Preliminaries

We review planning and their role in AC scenarios in this section.

2.1 Planning

A planning problem PP is a 4-tuple $\langle P, I, G, A \rangle$ where P is the set of predicates, I ($\subseteq P$) is the complete description of the initial state, G ($\subseteq P$) is the partial description of the goal state, and A is the set of executable (primitive) actions. A specification of an action consists of preconditions ($A_i^{pre} \subseteq P$) and postconditions ($A_i^{post} \subseteq P$). A plan for PP is an action sequence S, such that if S is executed in I, the resulting state of the world would contain G. A planner finds a plan by efficiently searching in the space of possible states configurations or action orderings (plans). It is desirable that a planner be *sound* and *complete*. A planner is sound if it will only generate correct plans. A planner is complete if it will always find a plan, provided one exists, given a domain and problem description. Automated planners are designed to be sound and complete.

A plan can be obtained without a planning algorithm and without explicit action specifications. An example of the former is when the user provides the plan directly and an example of the latter is when a plan is generated by some domain-dependent reasoning on initial and goal states. However, the soundness and the completeness of plan generator cannot be guaranteed. Domain-dependent planners usually produce superior plans than domain-independent methods, but they are harder to build and cannot be reused.

The user or system need not act on a plan immediately. A plan may be one of the many plans that are produced by users or planning algorithms before some plan is executed. They could be stored, searched, inspected, evaluated, modificated, and critiqued by human experts or automated reasoning systems, and executed. Eventually, plans will outlast their utility and be replaced.

2.2 Planning needs of Autonomic Computing Scenarios

In the AC vision[12], four aspects of self-management have been identified. We discuss the role of planning in these aspects.

Self-configuration: deals with installation, configuration and integration of IT systems. The installation procedures work by gathering information about the host environment, figuring out the dependencies among needed tasks and also optimizing performance measures, and finally executing the tasks to realize the changes. Information about host system is increasingly getting standardized along structured formats but the executable tasks can be ad-hoc scripts. Humans want to be closely involved in key decisions during execution.

Self-healing: deals with determination of problematic situations and recovering from them. It requires the system to reason with how activities can be performed, how diagnostic information is produced and how new changes can be affected with minimal cost and maximum benefit. The specification of actions could be known at some level of granularity.

Self-optimizing: deals with improving the performance of running systems by leveraging alternative opportunities. The system would monitor its performance and based on its changing environment, could initiate new changes (e.g., resource re-provisioning).

Self-protecting: deals with monitoring the environment for threats and responding to them. It is related to self-optimizing aspect but with the difference that the situation needs time-bound response and lead to cascading effect. Humans want to be closely involved based on the seriousness of the situation.

Type	I	G	A	S	Constraints
Self-configuring	Yes	Yes	-	-	Yes
Self-healing	Yes	Yes	Yes	-	Yes
Self-optimizing	-	-	-	Yes	Yes
Self-protecting	-	Yes	-	Yes	Yes

Table 1. The level to which planning problem information is expected to be available in AC scenarios. (-) means not assured.

Table 1 summarizes the level to which information about the initial state (I), goals (G), action specification (A), existing plans (S) and domain constraints (Constraints) is expected to be available in the different AC scenarios. In self-configuring situation, actions may be scripts whose pre- and post-condition information may not be known and there may be no plans available *a priori*. In self-healing scenario, A is expected so that alternative plans could be explored. In self-optimizing and self-protecting scenarios, a plan would be available for the running system but the goal specification will be more clearly defined for the latter.

From the above discussion, planning needs for AC can be summarized as follows:

1. The plan representation can be as general as workflows, e.g. BPEL4WS[6], with sequence, conditional, parallel, non-deterministic and loop constructs.

2. The plans are needed even if the initial state, goal state and action specification are not available, individually or collectively.

3. Automated plan generation is important but plans could also be obtained by users or domain-dependent methods. Even automatically generated plan may be analyzed by users before execution.

4. Over time, there would be a repository of previously generated and executed plans. They have to be considered while selecting existing or generating new plans.

5. The plans would typically be centrally executed but in large applications, the execution can be distributed.

3 Two Planning-based Systems for AC Scenarios

We discuss two sytems that use planning capabilities for AC scenarios. They will illustrate the potential of existing solutions and help in identifying domain-specific challenges for planning.

3.1 Domain-dependent Planning for Configuring Systems with CHAMPS

Configuring computing systems is an error-prone and costly step in setting up any IT solution. In this process, individual components are assembled and tuned to deploy a working solution. Typical problems include installing and configuring a multiple-machine deployment of J2EE based enterprise application along with its supporting middleware software like web servers (e.g., Apache Tomcat, IBM HTTPServer), application servers (e.g., IBM WebSphere) and databases (e.g., mySQL, IBM DB2). The aim is to provide intelligent support to system administrators and automate large parts of the configuration process such that the system could reconfigure itself when necessary. Plans originate from human experts similar to scripts capturing best practices or they are generated by techniques from planning, scheduling and domain-based dependency reasoning[11]. The plans are represented as BPEL4WS workflows analyze and are executed by human administrators, often with fully automated subplans, which the IT system takes care of without requiring human intervention.

An example of a complex IT application is the online book store application used in the Transaction Processing Council's Web (TPC-W) benchmark[18, 11]. It is a two-tiered application consisting of a web application server running 14 servlets and a database server working with 10 database tables. The servlets are hosted by a servlet container which depends on the web application server and operating system (OS). The database system also depends on the operating system, but the two application tiers can be on distinct machines and OSes. More details on domain-dependent optimized plan generation for change management and this application can be found in CHAMPS[11]. A plan to install the application would broadly consist of initalization (configuration information gathering) steps, two parallel sub-sequences for the two tiers and finally clean-up steps. Among the two sub-sequence, one is to set up OS, application server, servlet containers and servlets, and another is to set up OS, database server and database tables. There

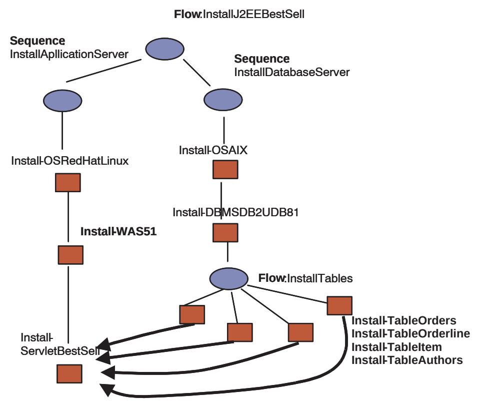

Figure 1. *A workflow plan to install BestSell servlet, a small part of the example bookstore application. Square represents basic/executable activities while oval represents structured activities (e.g., sequence). The arrows represent explicit synchronization dependencies between activities in the workflow.*

is dependency across the two sub-sequences as well. Figure 1 shows and example of a plan for installing one servlet (a small part) of the application which can be generated and executed by CHAMPS.

The advantage of a domain-dependent planner like CHAMPS for self-configuration is that it can find efficient plans for the specific scenario. However, this planning does not shed light on how planning in other AC scenarios could be covered. Another weakness, which is also common with all existing planners including ABLE as discussed later, is that plans are meant to be executed immediately. There is no support for storing, tracking and reusing plans.

3.2 Domain Independent Planning with ABLE

ABLE[1] 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.

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

ABLE has been extended with a planning rules that is compliant with the planning community's Planning Domain Description Language (PDDL[7])[15]. Since PDDL comes in various flavors, i.e. levels, the planning rules cannot be tied for processing to any specific planning engine. Therefore, it has a general planning framework called Planner4J comprising of a set of common interfaces[2], 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).

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. If a user already has the planning domain and problem information in PDDL, the planner can be invoked directly with minimal effort. 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.

A scenario for self-recovery of web applications was shown with ABLE in [15]. The objective is that a web application (e.g., the Online Book Store) should be able to automatically self-configure and self-heal in response to runtime exigencies to keep the website available. The problem had two machines and the web application consisted of two 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.

The problem was modeled as a planning problem with the initial state capturing the hardware and software considerations and the goal encoding the self-management objective. The set of actions explicate the preconditions and postconditions while initiating valid system response activities. The planner would find a plan to achieve the goal and then initiate (user-defined) Java methods corresponding to actions in the plan to materialize the overall system re-

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

sponse.

The key benefits for using the planning-enabled ABLE for AC scenarios 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 enhance and develop new planners faster by reusing existing components.
- The existing range of learning beans, rule types and data filters can be used to build complex planning agents.

However, ABLE is only making standard planners available for AC scenarios. The algorithms assume that the domain specification (P and A) is complete. Moreover, there is no support for storing, tracking and reusing plans.

4 AC Specific Challenges in using Planning

Based on our survey and experience of applying planning to autonomic computing, we identify two important challenges that AC applications pose to automated planning research – the need to support planning in partially specified domains, and the need to support plan life-cycle management. In this section, we describe these briefly.

4.1 Handling Incomplete Domain Model

The fact that a domain model is incomplete means many things. It could mean that domain is incompletely known though whatever is known is correct. This is orthogonal/different from expressiveness of domain model, e.g., PDDL levels[7], where the domain model at each level is complete though it may abstract some details of the world, which may get revealed in a more detailed higher PDDL level. Expressiveness impacts the complexity of planning and the representation of output plan. Incomplete domain model is also orthogonal/different from planning formulations varying in complexity like classical, conditional with partial observability, etc., where a problem is intrinsically of one type and hence cannot be expressed in any more simpler form.

If a plan is generated with incomplete domain models, it leads missing or under specified causal dependencies between actions in the plan. This affects the soundness guarantee of the planner because a generated plan may turn out to be not executable.

A domain may be incompletely specified in many ways. Formally, a planning domain is incomplete if at least one of the following happens (* denotes the corresponding complete specification):

- $P \subset P^*$
- $A_i^{pre} \subset A_i^{pre*}$ for some A_i
- $A_i^{post} \subset A_i^{post*}$ for some A_i
- There are relations $\alpha_i : \{P_i\} \times P$ which are not reflected in causal dependencies for achievement of predicates in A
- There are relations $\beta_i : \{A_i\} \times A$ which are not reflected in causal dependencies among actions in A

In the first case, the list of predicates in the domain is incomplete. In the second case, the list of preconditions for an action is incomplete. The preconditions can also be disjunctive but in contrast to traditional planning where disjunction is due to inherent uncertainity that will disappear at runtime, disjunction due to incomplete model will only get resolved with more domain input. In the third case, the list of postconditions of an action is incomplete and it can only get resolved with more domain input.

In many real domains like AC, the dependency among tasks or predicates is given but it is not explained in terms of a causal explanatin (i.e., what precondition/effect dependencies are violated if the dependency is violated). For example, it is known that a specific action must occur before another action but this information is known as an ordering relation ($A_i \prec A_j$) but the actions do not have a causal dependency in terms of the modeled pre- and post conditions [13]. The fourth and fifth cases represent specification of dependencies among predicates and actions, respectively, that do not have causal explanation. Axioms can be used to specify these types of incompleteness.

The challenge for planning community is how to effectively deal with such incompleteness. There has been some initial work, e.g., in[8], the authors look at the problem of evaluating plans when the underlying actions are incompletely modeled. They define four types of risk based on the structure of the plan provided that any action's specification can be corrected in future. Plans are compared based on their assessed risks, and a ranking is derived. To plan with relations that do not have causal dependencies, techniques from the intersection of planning and distributed scheduling (c.f. [2, 13]) will need to be adopted and extended.

4.2 Managing Life Cycle of Plans

A plan is synthesized for meeting some goals. But synthesis is just the beginning of a complex life-cycle management process. Plans must be organized in large collections, where they can be grouped along different purposes and are amenable to search, inspection, evaluation, and modification by human experts or automated reasoning systems. With users in the loop, plans which have been used in the past and have been successful, are more likely to be used again. New plans would get requested only when there is a deficiency in the existing plans. Eventually, plans will outlast their utility and be replaced.

Planning community has focussed primarily on synthesis. To support AC applications, one needs to manage the life cycle of plans within an application and based on the context of their usage. For example, one needs techniques to automatically generate metadata annotations of plans that could be used for storage and retrieval. If humans provide metadata, each annotation could be different and metadata mismatches will become a critical issue unless the user is very constrained.

The challenges in generating metadata for managing plans are many. The plan can be as expressive as general workflows with both automated and manual sub-plans. The specification of the pre- and postconditions of each action may not be available. Furthermore, the initial situation for which the plan was generated and the goal it is supposed to achieve are seldomly available. This lack of information, which is taken for granted in AI planning, neccessitates new techniques to deduce a plan's usage context. An initial approach for plan life cycle is discussed in [17] where plan analysis techniques take BPEL4WS workflows or PDDL plans as input, build action models using plan structure and generate metadata based on the given plan and as well as compared to other plans in a plan repository.

5 Relationship with other technologies

We now discuss relationship of planning with procedural policies, and Web services and Grid.

5.1 Relationship between procedural policies and plans

As mentioned in the introduction, the term policy is used to refer to any declarative specification of behavior that is desired from a software system but they usally refer to procedural policies. There are many choices for a procedural policy language for AC, e.g., WS-Policy[3] being defined for web services and REI[4]. Most languages support variations of the Event-Condition-Action (ECA) specification. ECA rules specify what actions to take in response to events provided stated conditions hold, i.e., [1]:

On : $\prec$ *Event* $\succ$
If : $\prec$ *Condition* $\succ$ *holds*
Do : $\prec$ *Actions* $\succ$

Action refers to any activity that can be performed in the domain and a policy may consist of one or more actions. The policy language may additionally allow specification of scoping and priority (business value) of rules that can be used for rule selection while working with a set of rules.

Planning can be used for managing procedural policies - while creating new policies, validating properties with exising policies, updating policies - based on whether a set of policies could be composed. In [14], it was shown how decision support problems in managing software components over their life cycle could be answered by posing them as planning problems. The same could be done for procedural policies. For example, suppose there are three policies related to managing data about a customer and sending out monthly report (see Figure 2). Policy-1 states that whenever a customer account is deleted, the account details are archived and the account is made inactive. Policy-2 states that when an account is archived, the customer is informed (to notify record retention requirements). Policy-3 governs monthly account report that is prepared for active accounts. Now if the policy designer wants to check if the monthly report will be sent to a customer whose account has been deleted, a planning problem can be made to answer whether a subset from the existing policies are composable to produce such a situation. The initial state will encode that the account is deleted, the goal state will encode that the account report is sent to the user, and the specification of planning actions can be derived from the specification of existing policies. It turns out that no plan is possible with the

[3]ftp://www6.software.ibm.com/software/developer/library/ws-policy.pdf

[4]http://ebiquity.umbc.edu/v2.1/get/a/publication/57.pdf

existing policies, and hence the stated situation is not possible.

```
Name: Policy-1
    Event: Customer-account-deleted
    Condition:
    Action: Account-archived, Account-inactived
Name: Policy-2
    Event: Account-archived
    Condition:
    Action: Send-customer-intimation
Name Policy-3
    Event: Month-end
    Condition: Account-active
    Action: Send-monthly-report

Policy situation to verify
    Event:
    Condition: Customer-account-deleted
    Action: Send-monthly-report
```

Figure 2. The set of exisiting policies and the policy to validate in the example to manage policies.

Reciprocally, procedural policies can be used while planning for the AC scenarios. Essentially, they allow users to decide what decision to make in a situation, and this information can be used to pick any information needed for planning. More specifically, procedural policies can be used to:

- Select information for goals (G)
- Select information for initial state (I)
- Select actions relevant for planning (A) and what gets modeled in their specification.
- Select predicates in the planning problem (P).

5.2 Relating AC with Web Services and Scientific Flows

Planning is actively being applied for composition of web services[16] and scientific workflows (grid)[4]. There are interesting similarities and contrasts between the planning requirements of autonomic computing and those of web services and scientific workflows. All of them require an expressive plan representation like BPEL4WS. All of these applications also pose the challenge of incomplete domain theories. In the case of web services, the incompleteness may come because of faulty or incomplete service annotations, while for workflows, the incompleteness may come because of constraints and dependencies without causal explanations. Plan management is also critical for these applications, while the need for automated synthesis is less prominent. In grid and web services, the plans will be distributedly executed while they will be primarily centrally executed in AC. Hence, techniques from distributed planning for generating concurrent plans are more relevant to the former.

6 Conclusion

In this paper, we explored the planning needs of AC, its match with existing planning technology, and its connections with policies and planning for web services and scientific workflows (grids). We observe that (1)automated planning technology is an evolutionary next step for AC systems that use procedural policies today and (2) AC requirements call for plan synthesis and management techniques that work with incomplete domain specifications (theories) and support a life cycle view of plans.

7 Acknowledgements

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

References

[1] Bailey, J., Poulovassilis, A., and Wood, P. 2002. An Event-Condition-Action Language for XML. *Proc. WWW, Honolulu, Hawaii.*

[2] Beck, C., Fox, M. Scheduling Alternative Activities. AAAI/IAAI 1999: 680-687.

[3] 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.*

[4] 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).*

[5] Blythe, J. 1999. An Overview of Planning Under Uncertainty. *AI Magazine, Vol. 20(2), pp. 35–54.*

[6] Curbera, F., and others. 2002. Business Process Execution Language for Web Services (BPEL4WS). *http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.*

[7] 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.*

[8] Garland, A. and Lesh, N. 2002. Plan evaluation with incomplete action descriptions. *In Proc. of the 18th National Conference on Artificial intelligence (AAAI'02), USA.*

[9] 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 Scheduling. Vol. 76*.

[10] 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*.

[11] Keller, A., Hellerstein, J., Wolf, J., Wu, K. and Krishnan, V. 2004. The CHAMPS System: Change Management with Planning and Scheduling. *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS 2004)*.

[12] Kephart, J. and Chess, D. 2003. The Vision of Autonomic Computing. *IEEE Computer, Vol. 36, No. 1, pp 41-50.*

[13] Munindar P. Singh, Greg Meredith, Christine Tomlinson, Paul C. Attie: An Event Algebra for Specifying and Scheduling Workflows. DASFAA 1995: 53-60.

[14] Srivastava, B. 2004. A Decision-support Framework for Component Reuse and Maintenance in Software Project Management. *IEEE 8th European Conference on Software Maintenance and Reengineering (CSMR 2004), Tampere, Finland.*

[15] Srivastava, B., Bigus, J., and Schlosnagle, D. 2004. Bringing Planning to Autonomic Applications with ABLE. *Proc. IEEE 1st International Conference on Autonomic Computing, New York, USA.*

[16] 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*.

[17] Srivastava, B., Vanhatalo, J., and Koehler, J. 2005. Managing the Life Cycle of Plans. *IBM Research Report.*

[18] TPP-Council. 2002. Transaction Processing Performance Council Benchmark W Specification (Web Commerce) v1.8. *http//www.tpc.org/tpcw.*

[19] Weld, D. 1999. Recent Advances in AI Planning. *AI Magazine*, Volume 20, No.2, pp 93-123.