Artur Andrzejak, Ulf Hermann, Akhil Sahai[1]

FeedbackFlow - An Adaptive Workflow Generator for System Management

[1]Hewlett-Packard Laboratories,1501 Page Mill Road, Palo Alto, CA 94034, USA

ZIB-Report 05-12 (Februar 2005)

FEEDBACKFLOW - An Adaptive Workflow Generator for System Management

Artur Andrzejak, Ulf Hermann
Zuse-Institute Berlin (ZIB)
Computer Science Research
Takustr. 7, 14195 Berlin, Germany
{andrzejak,hermann}@zib.de

Akhil Sahai
Hewlett-Packard Laboratories
Internet Systems and Storage Lab
1501 Page Mill Road, Palo Alto, CA 94034, USA
akhil.sahai@hp.com

Abstract

We describe a prototypical framework that further automates system management by composing complex management tasks from elementary actions, and executing composite tasks with feedback-awareness. FEEDBACKFLOW *implements a general closed control loop of* planning - execution - result validation - replanning, *and generates workflows of system management actions in an adaptive manner. System-dependent behaviour of the loop is specified by declarative description of the domain (essentially descriptions of available actions), and statement of the goal. We evaluate the design of this framework on examples taken from resource construction in Utility Computing environments, and discuss the challenges we have encountered. Our implementation utilizes external components such as* MBP, *a* PDDL*-conform planner, and* Triana, *a workflow specification and execution framework. An alternative approach involving* BPEL4WS *is discussed.*

1 Introduction

A large share of system management tasks can be broken down into small units, which can be described as *atomic actions*. Examples of such actions include installation or update of a software package from an image, including a new directory in the PATH environment variable, adding entries to the Windows registry, starting a standardized script, or rebooting a system. Understanding which preconditions are necessary for such an atomic action to succeed and which actions have to be taken in which order to achieve a desired goal (e.g. installation of a multi-tier application) currently requires human guidance. This analysis process consumes a significant fraction of the time of a human operator and is error-prone (more than 50% of system failures are caused by human operators [6]). Furthermore, when atomic actions are chained the results of each step must be verified manually, consuming more time, and a failure frequently requires the repetition of the analysis.

On the other hand, the pool of atomic actions, if properly parametrized, is largely similar on the thousands of systems with standard operating systems and basic software. This implies that the sequences of steps needed to reach a desired system state configuration are also alike. We believe that there is significant potential for automatizing system management tasks by exploiting these similarities. The idea is to create a pool of atomic and composite action descriptions common to many systems, which would save time of analysing their dependencies and allow automatic composition of such building blocks. Our larger goal is to automatically extract specifications of atomic actions from sets of management logs. The STRIDER system [22], which has partially inspired our work, shows that it is possible to identify Windows registry entries responsible for malfunctions from a large set of registry snapshots.

Assuming a sufficient pool of atomic action specifications (preconditions and effects) derived either manually or by means of machine learning, it is possible to automatically map action task graphs using planning algorithms. In this paper we focus on this approach to automation, and enhance it with the feature of automatic graph correction in response to partial failures.

1.1 Tackling complexity by reductionism

We adopt the view that most management tasks can be represented as a composition of parametrized atomic actions. Although this thesis has been not empirically verified, we believe that it is a reasonable assumption. Our project is an attempt to demonstrate this premise by simplifying the complexity of system state description and its changes sufficiently in order to reach an increased level of automatisation. Naturally, we have to assume certain properties of the system state (or, equivalently, the system configuration) for this demonstration. These simplifying assumptions are:

- *Containment* - system state can be “modularized”, i.e. we can consider the behavioral characteristics of each aspect or functionality of a system independent of other aspects. For example, the behaviour of the MySQL database (treated as a "functionality" of a system) does not depend on the behaviour or type of the file system. This is obviously a simplification, yet it is necessary to render the behaviour of this database comparable across many systems. Moreover, with this assumption the whole system state becomes a "Cartesian product" of the states of all modules, reducing its overall complexity.

- *Discretization* - the state of each part of the system can be modeled as a collection of attributes with small value sets. This is trivial for binary attributes, but numerical attributes like buffer size requires "quantization" into few discrete values, e.g. small, medium and large types. We assume that the specific system part behaves alike for all values within each range.

- *Measurability* - the values of such attributes (or the corresponding discrete values) can be determined from the system configuration in a standardized way. This process depends on the operating system or software, but it is not hard to implement, since the information is usually saved in clearly defined persistent stores or accessible via operating system interfaces. Examples include the Windows registry, environment variables on UNIX-like systems, settings files of individual applications (e.g. .emacs), entries of an LDAP-server in a Grid computing environment.

- *Locality* - the preconditions for execution of atomic actions can be tested by examining a small number of clearly defined parts of the system state ("modules"). Also the effects of such atomic actions influence a limited number of specified system state parts. For example, an action installing an application server only needs to test whether a specific Java version and a specific database is available, but does not need to know the type of the file system. Also, this installation action only influences a few registry entries on Windows systems or a few directories/environment variables on UNIX systems.

- *Atomicity* - a management action can be considered "atomic", i.e. it is not necessary to consider intermediate state changes in the planning process. This assumption can be enforced through subdivision a (non-atomic) action. An example is software installation comprised of installing several packages.

While it is not hard to find counterexamples to these assumptions in current systems, there are several management areas which allow this kind of simplification. These areas include Change and Configuration Management and Support (CCMS), resource construction in Utility Computing environments, software installation, and workflows in distributed environments, e.g. Grids.

1.2 Approach and results

Based on the above assumptions, we have built a prototypical framework featuring automatic planning of the action task graph, and the replanning of this graph in case of partial failures. FEEDBACKFLOW attempts to relieve the system operator from the need to analyse which actions are necessary to achieve a desired state, in which order they must be taken, and whether the execution of these actions was successful. Specifically, this is done by using AI planning techniques to obtain a task graph of necessary actions derived from a given set of (parametrized) atomic actions with corresponding preconditions and effects (additional input is the system state and the target statement). Such a task graph is subsequently represented in as a workflow for reasons of technical convenience such as debugging, visualisation, and reusability. During the execution of this workflow, feedback on success or failure of each atomic action is collected in order to update the system state and possibly repeat the execution with an adapted plan. The implementation uses the external components *MBP* [13], a *PDDL*-conform planner, and *Triana* [20], a workflow specification and execution framework. The details are described in Section 3.

A system prototype suffices to demonstrate the feasibility of the automation approach outlined above. It also provides an experimental framework to identify bottlenecks and the hard problems in the full realisation of this approach. The prototype also serves as the basis for an infrastructure for verifying whether the atomic action specifications are correctly captured from the examples by machine learning (a part of our future work mentioned above). By working with real-life examples we can examine which of the assumptions from Section 1.1 are tolerable in real systems, and which must be corrected. Finally, FEEDBACKFLOW can serve as a rudimentary testbed for studying the closed control-loop management approach proclaimed in the IBM's Autonomic Computing loop Monitor - Analyse - Plan - Execute [2].

2 Related Work

The field of automatic workflow generation is most closely related to our work. Automation solutions have been proposed in the contexts of Grid computing [7, 8], VLSI design tools [18], and business processes [4, 23, 17]. In [7] the automatic mapping of complex tasks to Grid resources is proposed. The starting point is a declarative description of available application components. In the first step, an

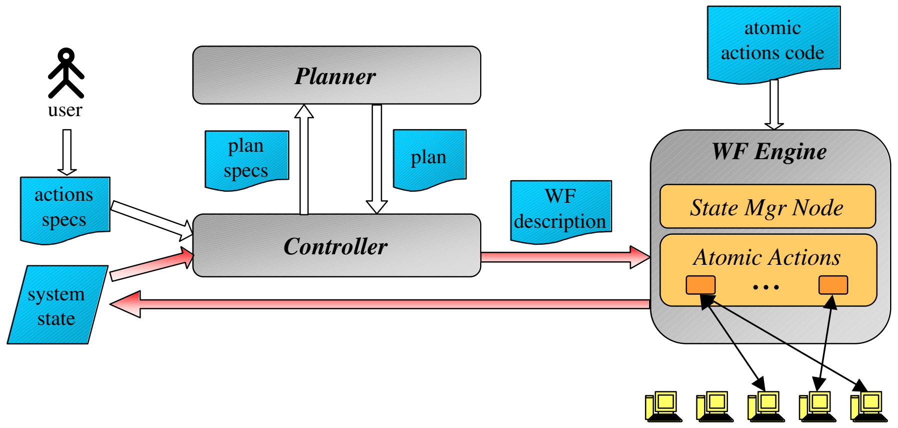

Figure 1. Architecture of FEEDBACKFLOW

Abstract Workflow is generated, which captures the workflow at a more abstract, functional level. Subsequent steps refine this result to a Concrete Workflow, which includes the detailed description of the execution step such as physical resource names and addresses, data transfers, etc. In contrast to our approach, [7] does not provide a mechanism for automatic replanning on execution failure.[8] discusses the same approach in the contexts of Grid Web Services, and workflow monitoring and maintenance on the Grid. In [18] a tool for generating workflows of VLSI design tasks is discussed. The tasks' data and functional dependencies are modeled by a graph. The authors describe a simple graph-theoretic algorithm used for generation of these specialised workflows. PLMFlow [23] uses a backward chaining algorithm to calculate a workflow for business process activities, which are defined by sets of predecessor and successor activities. The Oz Collaborative Workflow Environment [4] supports automation of assembling software products from components. It uses precondition and effects to model dependencies, and chaining algorithms for workflow generation. [17] discuss the use of AI planning techniques in the domain of business processes. Support tools for software installation might be also considered in this category. For example, the Gentoo-Linux tool Portage [10] considers dependencies between software packages, and installs the required packages together with the target software.

Policy-based systems like BMC Patrol [5] employ event-condition-action rules, but have the disadvantage of high complexity of the rule specification and debugging. More elaborate approaches involve case-based reasoning [21], which however suffers from high computational complexity. Policy languages have also been utilized for automated construction of virtual server farms [16] in utility environments. The authors use CIM as the underlying resource model and a constraint satisfaction problem solver to generate a description of a composite resource.

Within the field of Autonomic Computing, feedback-based approaches have been applied to managing QoS and performance of storage systems. Clockwork [15] is a prototypical self-tuning framework for load balancing in a network attached storage system. It uses feedforward control in predicting demand by means of an ARIMA model. Another category of model-based approaches includes Minerva [1], which derive management decisions from detailed models of the storage systems.

Planning is a field of Artificial Intelligence that evolved from simple first-order predicate logic problem solvers [19] to solvers that can manage complex problems involving temporal logic conditions. Although current planning algorithms are still not efficient enough to handle large instances, essential progress has been made in mapping planning problems onto satisfiability and model checking problems. Along with this development, the yearly International Planning Competitions has established the Planning Domain Definition Language (PDDL) as a common specification standard [9].

3 Architecture

FEEDBACKFLOW attempts to automate creation and execution of workflows for system management in an adaptive way. As input it accepts a declarative specification of a collection of available atomic actions (such as software installation, transfer of a file or directory, commando execution), a description of the current system state of the managed system, and a declarative specification of a target system state. Based on this input, FEEDBACKFLOW generates a workflow of the (parametrized) atomic actions, starts a workflow execution engine (which enforces the execution of each atomic action), and compiles the execution results into an updated system state. If the updated system state indicates that the system has not been able to reach the target system state, a new execution cycle is initiated, with a new workflow based on the updated state information. This control loop (illustrated with filled arrows in Figure 1) is executed until the target system state is reached.

The architecture of the system consists of the following active components (Figure 1):

- a custom *Controller*, which triggers the consecutive steps in the control loop and translates the input/output between components
- a PDDL 2.1-conformant *planner* (currently MBP)
- a *workflow execution engine* (currently Triana)
- for each of the managed hosts, a *daemon* which receives parametrized descriptions for local atomic actions and executes them (currently a combination of ssh and Ant).

The interaction between these components involves a number of intermediary document types. The most important of these are the three types which must be provided by the user:

1. A specification of atomic actions available in the system (e.g. execution of a "dumb" Ant script), together with a statement of a target of the workflow (e.g. instantiation of a multi-tier server farm), both in PDDL syntax (see Section 3.3 for details).
2. A specification of the system state in PDDL syntax, which is basically a list of identifiers representing resources together with the assignments of attribute values.
3. A collection of Java class/Ant script pairs, each implementing a different type of an atomic action (see Section 3.1).

3.1 Planning and Action Execution

Interaction between the components in FEEDBACKFLOW is orchestrated by the Controller as shown in Figure 1. Upon receipt of an actions/target specification, the Controller generates a plan specification, and sends this document to an (external) planner. We currently use MBP, as this planner supports the largest set of features among the publicly available packages. If the plan cannot be generated or the planner did not finish before a deadline, an error is reported and the execution finishes. This scenario usually results from an inadequate action specification (either description errors or an intractable complexity level) and requires the user to correct the input.

In the normal case, the generated plan is parsed by the Controller, which translates it into an XML-description of a Triana workflow. The generated workflow consist of a collection of atomic actions and a so-called *(System) State Manager* described in Section 3.2. Each atomic action is implemented by a Java class provided by the user. The Java class is responsible for communicating with the daemon at a managed resource in order to trigger the execution of an Ant script (residing locally on the resource), collect the results of the execution, and report it to the State Manager. In addition to this proxy-like role, a Java class can be put into simulation/debugging mode, so that it does not communicate with a remote demon, but only responds to a trigger with a randomly generated "success/failure" result.

The resource-local Ant script executes a specific action such as software installation, file transfer, compilation etc. according to the parameters received via the daemon call. It should be clear that each Java class/Ant script pair is quite generic and parametrized, e.g. we would need only one Java class/Ant script to handle most file transfer operations, and the pair can be reused in different application scenarios. Similarity among Java classes (differences are mostly due to the number parameters to be passed to the Ant script) also allow us to use inheritance to a large degree, and in most cases only a few changes are necessary to create a new Ant script to accompany the Java class.

3.2 Workflow

The Controller triggers the planner to produce a sequence of actions with appropriate parameters. The Controller provides the planner with action specifications (Figure 3), the system state description, and the target statement (Figure 4). The output of the planner is transformed by the Controller into a description of a Triana workflow, as depicted in Figure 2. Such a workflow consists of a collection of units, which can be executed either in parallel or sequentially. Each unit in Triana has zero or more data inputs, zero or more data outputs, and is fired upon reception of data on one or all of its inputs [20]. In this case a Java class corresponding to this unit (and provided by the user) is instantiated and executed.

The workflow generated by FEEDBACKFLOW consists of a single State Manager unit, a Stop unit and a collection of units representing the atomic actions. The State Manager maintains a view of the current system state as it is changed by the execution of the atomic actions, i.e. it collects feed-

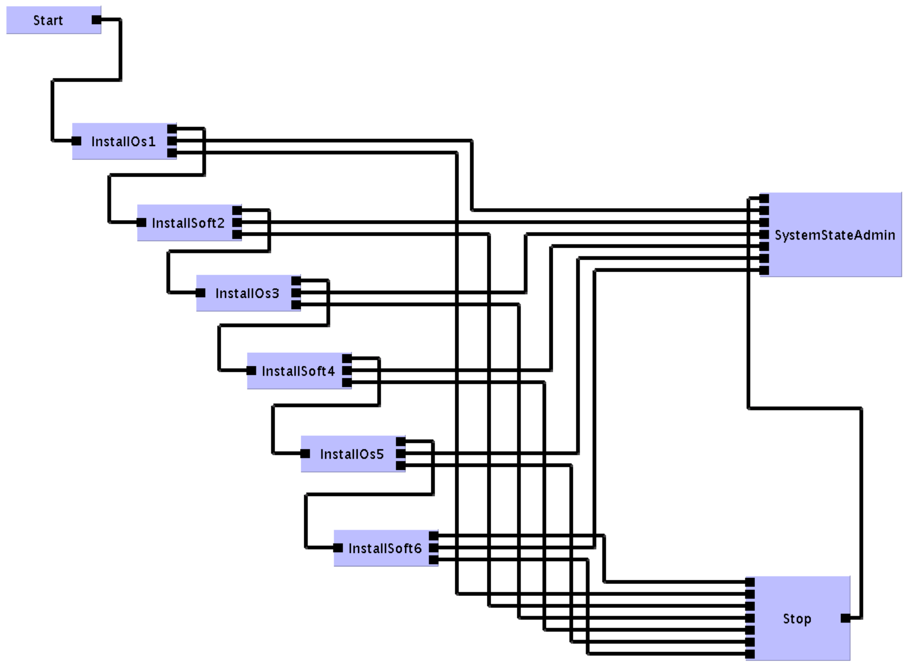

Figure 2. An example of an automatically generated workflow

```
(define (domain installWorld)
(:types int os soft host)
(:predicates
    (isAvailable  ?s - soft ?o - os)
)
(:functions
    (hasOs    ?h - host) - os
    (hasSoft  ?h - host) - soft
    (notInstallOS ?h - host ?os - os) - int
    (notInstallSoft ?h - host ?s - soft) - int
)
(:action InstallOs
    :parameters (?h - host ?o - os)
    :precondition (and
                    (= (hasOs ?h) noOs)
                    (= (notInstallOS ?h ?o) 0))
    :effect (assign (hasOs ?h)  ?o)
)
(:action InstallSoft
    :parameters (?h - host ?s - soft)
    :precondition
        (and
            (= (notInstallSoft ?h ?s) 0)
            (= (hasSoft ?h) noSoft)
            (exists (?o - os)
                  (and
                      (isAvailable ?s ?o)
                      (= (hasOs ?h) ?o))))
    :effect (assign (hasSoft ?h) ?s)
))
```

Figure 3. An example PDDL file specifying software installation actions

```
(define (problem install)
(:domain installWorld)
(:typedef int - (range 0 3))
(:objects
    apache - soft  mysql - soft  tomcat - soft
    noSoft - soft
    winxp - os  linux - os  noOs - os
    host1 - host  host2 - host  host3 - host
)
(:init
    (isAvailable apache winxp)
    (isAvailable tomcat winxp)
    (isAvailable tomcat linux)
    (isAvailable mysql linux)
    (forall (?h - host)(= (hasOs ?h) noOs))
    (forall (?h - host)(= (hasSoft ?h) noSoft))
    (forall (?h - host)
        (forall (?os - os)
         (= (notInstallOS ?h ?os) 0)))
)
;; <begin-feedback-settings>
;; <end-feedback-settings>
(:conformantgoal (and
                    (= (hasSoft host1) apache)
                    (= (hasSoft host2) mysql)
                    (= (hasSoft host3) tomcat))
))
```

Figure 4. An example system state specification in PDDL

back from the atomic action nodes concerning the state of the resource after executing the action (e.g. success or failure). The Stop unit is executed only if an unrecoverable error has occurred (such as a failure of action execution in a linear sequence).

If several actions are to be executed in a sequence, the corresponding workflow will have a chain of the appropriate atomic actions. Additionally, each action's output is wired to the State Manager. In this way, finishing the execution of one action will simultaneously cause the firing of the next one and provide feedback to the State Manager. Upon completion of all action chains or upon an unrecoverable error signaled by the Stop unit, the State Manager updates the system state document, and workflow execution is finished. From then on the Controller assumes the execution, and either restarts the whole cycle or terminates it depending on the most recent system state.

As an alternative to the current solution, a plan received from the planner could be executed directly from the Controller or translated into a simple script e.g. an Ant script. However, having an intermediate Triana workflow offers some advantages. Those include: lower implementation complexity by decoupling the Controller from workflow execution, the possibility of re-using planned workflows as "templates" or combining them with manually created ones, and the features of the Triana workflow IDE for debugging and visualisation. Moreover, Triana greatly facilitates wrapping individual actions into Web Services. These advantages justify the increased heterogeneity of the whole system due to this architectural choice.

3.3 Specifying Actions and the System State in PDDL 2.1

As a specification language for atomic actions we use PDDL 2.1, a declarative language developed for international planning competitions [9]. The advantage of this choice is that most planning software can directly process this input, and PDDL is a de facto standard. A possible alternative would be XPDDL, an XML form of PDDL, which however has not reached maturity as of the time of writing [11]. We have also experimented with a proprietary XML-based specification language which was translated into PDDL by the Controller, but we negated this option because it was non-standard.

Figure 3 shows a simplified specification of software installation actions. The declared abstract types int, os, $soft$, $host$ are used in the subsequent predicate and function definitions. For example, the predicate $isAvailable$ applies if and only if the software $?s$ is available for the operating system $?o$ (arguments are prefixed with ? in PDDL). There are special functions $notInstallOS$ and $notInstallSoft$ called *prohibitors,* which are used to prohibit the corresponding action from being executed for a particular argument value combination.

Two atomic actions are indicated. The first one, $InstallOs$ can install the operating system $?o$ on the host $?h$ if this host has no operating system yet (expressed by the fact that the value of the function $hasOs$ equals the constant $noOs$) and the prohibitor for these arguments is disabled (has value 0). The effect of the action is the assignment $hasOs(?h) :=?o$. The atomic action $InstallSoft$ is only executed if the prohibitor is disabled, the host $?h$ has no installed software (a simplification to make the example clearer), and the host has an operating system for which the software is available. As an effect of this action, the assignment $hasSoft(?h) :=?s$ is performed.

Figure 4 shows an initial system state specification and the target specification (as provided by the user). After declaring constants (representing resources and attribute values) in the $objects$ section, the values of the predicates and functions are set. In the following section (delimited by *<begin-feedback-settings>* and *<end-feedback-settings>)* the modifications of the system state resulting from workflow execution are automatically inserted by the State Manager (Section 3.1). These modifications are represented by additional assignments of function and predicate values (in this example by setting the prohibitors to non-zero values for certain arguments).

The goal of this plan is specified in the last section of Figure 4. Here we require that the software $apache$, $mysql$ and $tomcat$ be installed on the hosts $host1$, $host2$, and $host3$, respectively.

3.4 Application to complex resource construction

In order to validate our system we have used examples from the domain of resource construction in Utility Computing environments. A typical task was to assemble a multi-tier system, where each tier required several servers of the same type and running the same application. Different applications were available for different operating systems or sets of them. Further constraints involved a limit on the total cost of the system (cost was modeled as a function of resources).

Our approach worked well for small examples, but we encountered a critical complexity limit, depending on the number of used "objects" (resources) and the number of "exists" operators. Beyond this limit, the MBP software took prohibitively long to compute a plan (we cut off a computation on a 1 GHz Pentium IV machine if it took longer than 4 hours) and consumed a large amount of memory. Unfortunately, we could not obtain any further information about this phenomenon from the authors of MBP. Further issues arose in modeling more complex conditions, such as an upper bound on the sum of costs. This required intricate expressions in PDDL, which makes this approach too complex for non-specialists.

Our experience with deploying FEEDBACKFLOW showed that the most time-intensive aspect of the system is the specification of the actions, i.e. identifying and encoding their preconditions and effects. Frequently not all relevant characteristics can be captured in the first attempt, which makes debugging necessary. Although our approach facilitates the specification process by using a declarative description language, this stage is still a bottleneck in automatizing tasks. We believe that methods for automatic deduction of the action specifications (such as Inductive Logic Programming [14]) from collected real-world examples could bring some relief to this problem.

4 Design Alternatives

As a precursor to FeedbackFlow we attempted to enhance BPEL4WS [3] with declarative target descriptions and support for automatic (re-)planning. Since BPEL4WS is a de facto standard for business workflow descriptions, the design of this precursor system was guided by backward-compatibility to BPEL4WS. By adding dynamic replanning we could gain the following features:

- the possibility to mix "standard" (i.e. non-adaptive) BPEL4WS-workflow descriptions with the adaptive workflows in a single document,
- the ability to dynamic replan (part of) a workflow triggered and controlled from within a BPEL4WS-document.

We think that it is instructive to sketch this approach, even if we have abandoned it for the reasons stated below.

For the BPEL4WS system we extended a standard BPEL4WS-compliant description by adding sections for automatic workflow planning. Such a section was encompassed by a BPEL4WS activity of type `scope` with an additional attribute `plannedWorkflow="true"`. Inside this section there was a set of atomic activities represented by BPEL4WS activities of arbitrary type. For each activity the added tags `<pre>..</pre>` and `<post>..</post>` specified pre- and postconditions. To capture the current system state we used a (complex) BPEL4WS-compliant variable `SysState`. The pre- and postconditions refer to the components of this variable.

Upon entering such a section, the BPEL4WS-engine invoked an external Web Service `WorkflowPlanner` acting as a workflow planner. The engine passed the target system state and the current value of `SysState` to the planner, then looped awaiting messages from the planner. Such a message could initiate the execution of one of the actions, or terminate the loop, so that the automatic workflow planning section was left. Notice that all activities were expressed in a BPEL4WS-compliant language (we use the extensibility of BPEL4WS by introducing tags from a proprietary namespace) and were executed by a standard engine. The whole additional implementation and algorithms are hidden in the `WorkflowPlanner` service.

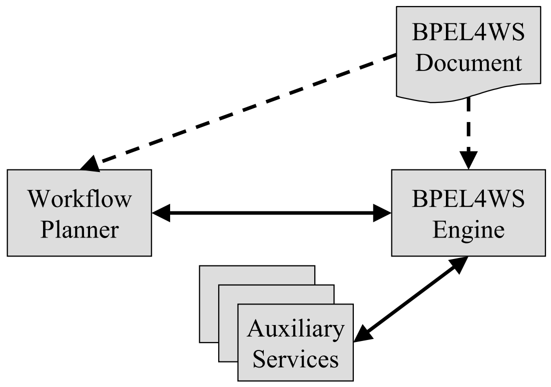

Figure 5. Interaction of system components

Figure 5 gives an overview of the system components' interaction. At the deployment stage, the complete workflow description was made available to the engine and the workflow planner. Once all services were running, and the engine entered an automatic workflow planning section, it called the workflow planner, and the interaction described above took place. Such an iteration could repeat each time when an automatic workflow planning section was about to be executed. As in FEEDBACKFLOW, the execution of this section could be repeated until a desired target system state is reached.

We abandoned this approach due to problems with the currently available non-commercial implementation of BPEL4WS [12] and the limited variable manipulation power of BPEL4WS 1.1. Regarding the first problem, the IBM AlphaWorks implementation of BPEL4WS does not support all of the features of the BPEL4WS standard, and even the included features do not work reliably. The latter problem relates to the fact that even simple manipulations of "arrays" (sequences of simple WSDL types) is impossible in the current version of BPEL4WS. This makes it necessary to delegate such processing to external web services, which increases both the system complexity and the cost of changes.

5 Conclusions

We have described a prototypical system for adaptive generation and execution of workflows in the domain of systems management. The kernel of FEEDBACKFLOW is a closed control loop comprised of state-aware workflow planning, workflow execution, generation of an updated system state, and re-iteration of this process until the desired system state is reached.

One key feature of our system is the description of available actions with the standardized, declarative planning language PDDL. This declarative specification is a prerequisite for flexible and adaptive workflows, in that it avoids the need to anticipate every possible situation and code a solution to it. Using a standardized and generic language such as PDDL allows us to exploit different external planners and to enhance the expressiveness of the input specification without changing the system. We note that this language is not constrained to the domain of systems management, so that by adding appropriate atomic actions our system can be easily adapted to handle problems from other domains as well.

The utilization of workflows as in intermediate stage in the execution cycle (as opposed to a direct execution of a plan) has a number of advantages. In addition to facilitating debugging and visualisation of the generated action sequences, it enables reuse of generated task sequences. Furthermore, the Triana framework is also able to transform individual actions into instances of Web Services.

Practical evaluation of our approach on examples from resource construction revealed two bottlenecks: the need for more efficient planning for larger problem sizes and the necessity of augmenting the action specification process. Our further work will address both problems. We intend to write a specialized planner which can handle a dialect of PDDL that scales better. The PDDL language will be modified by introducing special predicates (e.g. *contains*) and special classes of functions. Specifications using only these elements will be handled very efficiently through backward-chaining techniques, while other specifications will be recognized and sent to a general-purpose planner.

Simplifying and increasing the efficacy of the action specification process requires longer-term research and is inherently intricate, like the problem of specifying software requirements. Among the possible approaches we want to focus on automatic or semi-automatic discovery of action preconditions and effects from large sets of examples (given e.g. by examining the registry on Windows machines). The main tools here are existing and possibly adapted methods from Inductive Logic Programming [14] such as the FOIL algorithm. The output of these algorithms are rules in first order logic, which makes them easily convertible into a declarative language such as PDDL.

Further future work will be targeted toward extraction of the system state or configuration from persistent stores and conversion into our system state model.

6 Acknowledgments

We would like to thank Minor Gordon, ZIB Berlin, for help with editing this manuscript.

References

[1] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, 19(4):483–518, 2001.

[2] J. O. K. andDavid M. Chess. The vision of autonomic computing. *IEEE Computer*, (1):41–50, January 2004.

[3] BEA, IBM, Microsoft, SAP AG and Siebel Systems. *Specification: Business Process Execution Language for Web Services Version 1.1*, 2003.

[4] I. Ben-Shaul, G. T. Heineman, S. S. Popovich, P. D. Skopp, A. Z. Tong, and G. Valetto. Integrating groupware and process technologies in the oz environment. In *International Software Process Workshop (ISPW)*, pages 114–116, 1994.

[5] BMC Software. *Patrol for Storage Networking*, 2003.

[6] G. Candea, A. B. Brown, A. Fox, and D. Patterson. Recovery-oriented computing: Building multitier dependability. *IEEE Computer*, (11):60–67, November 2004.

[7] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny. Pegasus: Mapping scientific workflows onto the grid. In *European Across Grids Conference*, pages 11–20, 2004.

[8] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, K. Blackburn, A. Lazzarini, A. Arbree, R. Cavanaugh, and S. Koranda. Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing*, 1(1):25–39, 2003.

[9] M. Fox and D. Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61, 2003.

[10] Gentoo. *Portage, Gentoo's Software Management Tool*, 2004.

[11] J. Gough. *XPDDL: the eXtensible Planning Domain Definition Language, V0.1b*, 2004.

[12] IBM, AlphaWorks. *IBM Business Process Execution Language for Web Services JavaTM Run Time (BPWS4J)*, 2004.

[13] ITC-IRST, Italy. *MBP: a Model Based Planner*, 2004.

[14] N. Lavrač and S. Džeroski. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, New York, 1994.

[15] L. W. Russell, S. P. Morgan, and E. G. Chron. Clockwork: A new movement in autonomic systems. *IBM Systems Journal*, 42(1):77–84, 2003.

[16] A. Sahai, S. Singhal, R. Joshi, and V. Machiraju. Policy based resource construction in utility computing environments. In *IEEE/IFIP NOMS*, 2005.

[17] H. Schuschel and M. Weske. Integrated workflow planning and coordination. In *Database and Expert Systems Applications (DEXA)*, pages 771–781, 2003.

[18] V. Shepelev and S. Director. Automatic workflow generation. In *European Design Automation Conference, Euro-DAC*, 1996.

[19] Stanford University. *STRIPS - Stanford Research Institute Problem Solver*, 1972.

[20] I. Taylor, M. Shields, I. Wang, and R. Philp. Grid enabling applications using triana. In *Workshop on Grid Applications and Programming Tools*, Seattle, 2003.

[21] D. Verma and S. Calo. Goal oriented policy determination. In *Proc. 1st Workshop on Algorithms and Architectures for Self-Managing Systems*, pages 1–6, San Diego, CA, 2003.

[22] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, C. Yuan, and Z. Zhang. Strider: A black-box, state-based approach to change and configuration management and support. In *USENIX Large Installation Systems Administration Conference (LISA)*, pages 159–172, 2003.

[23] L. Zeng, D. Flaxer, H. Chang, and J.-J. Jeng. Plm $_{\text{flow}}$-dynamic business process composition and execution by rule inference. In *Technologies for E-Services (TES)*, pages 141–150, 2002.