# Managing the Life Cycle of Plans

**Biplav Srivastava**
IBM India Research Laboratory
Block 1, IIT, New Delhi 110016, India
sbiplav@in.ibm.com

**Jussi Vanhatalo** **Jana Koehler**
IBM Zurich Research Laboratory
CH-8803 Rüschlikon, Switzerland
{juv,koe}@zurich.ibm.com

### Abstract

The scalability of recent planning algorithms allows developers to automate planning tasks, which so far have been reserved to humans. However in real-world applications, synthesizing a plan 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 the search, inspection, evaluation, and modification by human experts or automated reasoning systems. Eventually, plans will outlast their utility and be replaced.

We present our solution to plan life cycle management for an autonomic computing application. We focus in particular on the automatic synthesis of plan metadata for plans containing conditional and parallel actions, well-structured loops, and non-deterministic choices. The plans are of unknown origin, i.e., their underlying action model, which could provide us with pre- and postconditions, is not available. New analysis techniques are presented that uniformly generate metadata for plans, thus allowing a system to embed plans into context and organize them in meaningfully structured plan repositories.

## Introduction

Configuring computing systems is an error-prone and costly step in setting up an IT solution. Individual components must be assembled and tuned to deploy a working solution. The main driving force behind our work is to provide intelligent support to system administrators and automate large parts of the configuration process with the goal of allowing an autonomic computing system to reconfigure itself. Plans originate from human experts similar to scripts capturing best practices or are generated by automatic planning systems responding to system requirements. Techniques from AI planning, scheduling and domain-based dependency reasoning can be used to generate, analyze, and execute configuration change management (CM) plans (Keller *et al.* 2004). Figure 1 introduces a CM system called CHAMPS, which applies these techniques to support system administrators in their complex activities. In a typical scenario, the system administrator will enter a change request into the CHAMPS system. From the request, a query is generated that is executed over a repository of plans, which have been enriched with metadata. These metadata can be provided by humans or are automatically generated by a plan analysis process we describe in more detail in this paper. If the query succeeds, one or more candidate plans are sent to a scheduling component, which will produce a change plan. If the query fails, the planner is activated to find a new change plan. The change plan is either executed automatically or it is proposed to the system administrator. The mode of execution depends on the information contained in the query, the change plans retrieved and their metadata information. Plans that require no human intervention and perform non-critical updates of an IT system that do not require human confirmation are executed automatically. All other plans require approval by the system administrator before they are executed.



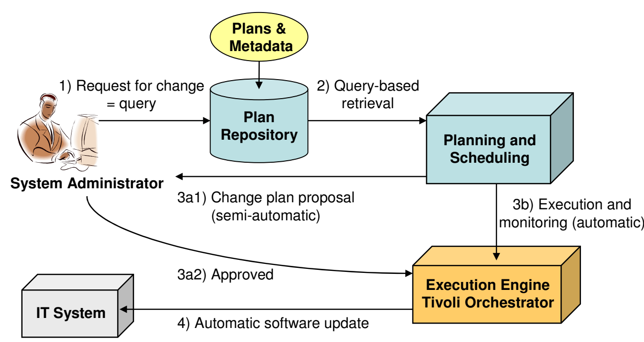


Figure 1: Change management using AI Planning.

To successfully run the application, a large set of CM plans needs to be managed. In particular, the plans need to be organized for storage and retrieval and visualized for human inspection and execution. A human administrator needs to know how a plan performed during previous executions, why it may be suitable in the current situation, how it relates to other plans available in the same domain, and whether the plan has critical features that may lead to execution bottlenecks. Such information is not contained in the plan itself and can only be derived from associated metadata. The uniformity of the metadata is critical for the quality of change management and should be provided by automatic plan analysis methods. Generating metadata by hand is a very costly and time-consuming activity and can easily lead to inconsistent and unusable information.

So far, there has been very little research on generating metadata for plans. A tool is presented in (Kim, Gil, & Spraragen 2004) that interactively works with a user to validate sequential workflows. The analysis determines plan properties such as whether a plan is *justified*, *satisfied*, or *correct*. The procedures resemble plan-pruning criteria during flaw refinement steps in partial-order planning and ensure that a correct workflow has all the desirable properties including that it is acyclic. The tool has access to the action specifications and is integrated with the means-end planner Prodigy to complete partially sketched plans. One of the first approaches that considers plans independently of plan synthesis is (Pollack 1992), where the author talks about the uses of plans, once they are obtained. The work on plan modification described in (Kambhampati 1990) classifies techniques to compute an explanation of plan correctness and to make the rationale behind planning decisions explicit. (Myers 2004) generalizes plans as relations among planning elements, which are actions, postconditions, and parameters. The idea is to generalize causal links so that qualitative relationships can be more easily communicated from/to users. (Garland & Lesh 2002) 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 the future. Plans are compared based on their assessed risks, and a ranking is derived. A database for PDDL-2 plan fragments is described in (Yaman *et al.* 2004) with the focus on maintaining consistency of the plan database. The database uses a query language based on a plan-specific relational algebra, whereas we use an expressive object-oriented query language in our solution. Model-checking techniques for the verification of workflows are investigated in (Fu, Bultan, & Su 2004).

Our solution differs from and extends previous work in the following way: Our metadata not only extend the metrics defined in (Kim, Gil, & Spraragen 2004), but are also computed for expressive workflows represented in BPEL4WS (Curbera & others 2002), which can contain loops as well as conditional and parallel branches. Our methods are targeted at managing collections of plans when the specification of actions as well as the planning technique are not available. We also analyze plans by comparing them with other available plans and based on their performance in previous executions. Such executions can be gathered from real-world system runs or from simulations that consider different execution conditions. Based on their associated metadata, plans are stored in a repository similar to a case base from where they can be retrieved in a very flexible way.

The plan analyzer and repository are currently made available inside IBM as part of the CHAMPS toolkit for CM.The plan analyzer will be made available externally as part of the Java-based programmatic planning framework *Planner4J* (Srivastava 2004), which also implements a variety of planners in Java sharing a common infrastructure. The *Planner4J* features are provided through the publicly available ABLE agent building toolkit[1](Srivastava, Bigus, & Schlosnagle 2004) and are being employed in many applications requiring diverse planning capabilities. The plan repository is available on request as an Eclipse plugin from the authors.

## Change Management Workflows

Change management requires complex activities, which do not map directly to simple sequences of actions. A workflow representation seems to be more appropriate and also facilitates the automatic or semi-automatic execution of the plan. A typical example is the installation and configuration of a multiple-machine deployment of a J2EE-based enterprise application along with its supporting middleware software such as web servers, application servers, and databases. We discuss an online bookstore application from the Transaction Processing Council's Web (TPC-W) benchmark (TPP-Council 2002), which 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 the operating system. The database system also depends on the operating system. The two application tiers can be on distinct machines and operating systems. Several plans are needed to install this application. First, a plan comprising configuration information gathering steps must be executed. It is followed by two plans, which run in parallel and install the various components of the two tiers. Finally, clean-up steps are needed. The subplan installing the web application tier sets up the operating system, application server, servlet containers, and servlets. The subplan to install the database tier sets up the operating system, database server, and database tables. Actions in one subplan can depend on the successful execution of actions in the other subplan.

In the CHAMPS system, plans are represented in the workflow language BPEL4WS (Curbera & others 2002). A BPEL4WS workflow is a specification of a coordinated set of activities where the activities may themselves be automated or manual. The specification contains information about partners and their roles, message types, variables, and activities. Two categories of activities are distinguished—basic and structured activities. Basic activities are executable and correspond to the actions in a plan. BPEL4WS has activities to *invoke* a web service, *receive* a message from another service, *reply* to a service, *throw* an exception, *terminate* the execution of a flow, *wait* for an event to happen, or simply to do *nothing*. Structured activities group basic activities within a *sequence*, a conditional *switch*, a *while* iteration, a non-deterministic choice called *pick*, or a parallel *flow*. With these constructs, the control and data flow of the workflow are precisely specified.

Figure 2 shows an example of a BPEL4WS workflow, which we will use throughout the paper. It installs the single servlet *Install-ServletBestSell*. The workflow contains a top-level flow into which two sequences are embedded that run in parallel inside the flow. The upper sequence installs

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

the application server. It contains three basic activities installing the Linux operating system, the Websphere Application Server (WAS), and finally the desired servlet. The lower sequence installs the database server. It contains two basic activities installing the AIX operating system and the DB2 database followed by another flow, in which four basic activities are embedded that install database tables, possibly in parallel. The dashed arrows denote explicit synchronization dependencies between activities in BPEL4WS. In the example, the database tables must be installed before the servlet installation can start, which is the last activity of the upper sequence. All basic activities are of the type *invoke* and represented as squares. They describe executable system scripts.

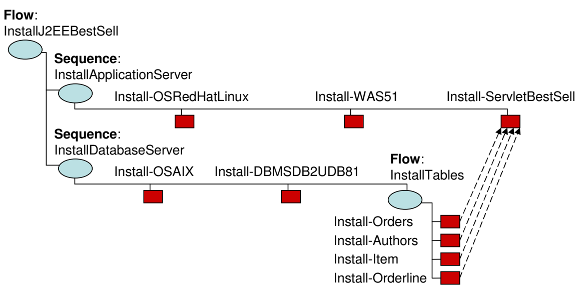


Figure 2: A workflow to install the *BestSell* servlet.

## Analysis of Plans without Explicit Action Model

We developed a comprehensive plan analysis method that takes BPEL4WS workflows or PDDL plans as input and generates an output report summarizing the analysis results. Results contained in the report are further processed to generate metadata of various formats that are associated with the workflow or plan. The workflows can originate from humans, i.e., a semantic action model 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 does not occur in AI planning solutions, poses particular challenges for the plan analysis.

*Function*: **AnalyzePlan**
*Input*: $P$ - Plan or Workflow
*Optional Inputs*: $A$ - Action Specification, $R$ - Plan Repository
*Output*: $O$ - Output Report
*Optional Output:* $A$ - (Derived) Action Specification

If $P$ is a workflow
    $P$ = ConvertWorkflowToPlanStructure($P$)
    (Additional output) $L$ - Set of all links in $P$
End-if
If $A$ is not specified
    $A$ = DeriveActivitySpecifications($P$, $L$)
End-if
$O$ = AnalyzePlanInternal($P$, $A$, $R$)
Return $O$ {Optionally $A$}

Figure 3: Top-level procedure for plan analysis.

Figure 3 summarizes the main steps performed during plan analysis. If the plan analysis is invoked with a BPEL4WS workflow, the plan structure underlying the workflow is extracted first using the procedure *ConvertWorkflowToPlanStructure()*. *AnalyzePlan()* can leverage existing specification of actions ($A$) in the plan, but it does not require them. If $A$ is not available, *DeriveActivitySpecifications()* determines pre- and postconditions of the actions based on ordering constraints between the activities contained in the BPEL4WS workflow and contextual information that is available from the semantic model of BPEL4WS. In *AnalyzePlanInternal()*, the syntactic properties of the plan structure are analyzed further. The analysis can also comprise a comparison of the plan, $P$, with previously built and analyzed plans from a given repository of plans ($R$). The output is an analysis report that reveals hidden properties of plans, while keeping the input plan unchanged. In the following subsections, the various phases of the plan analysis are explained in more detail.

### Extracting the Plan Structure

In PDDL, plans are represented as a sequence of time steps, where each time step can have parallel actions. This basic plan structure has been formalized as an IPlanSolution plan interface in (Srivastava 2004) and found to be common while representing different types of plans such as contingent, metric, and hierarchical plans. We adopt this plan structure for analysis, but in addition distinguish different types of actions that correspond to the activity types in BPEL4WS.

The procedure *ConvertWorkflowToPlanStructure()* takes the root of the parsed workflow fragment and creates the plan structure. Figure 4 shows the IPlanSolution structure for the example of Figure 2. Each activity at the same level of nesting in the workflow occurs in the same time step. The activities that are nested directly inside a flow all occur in the same time step following the step containing the flow. Activities that are nested inside a sequence are mapped to subsequent time steps.

0: INSTALLJ2EEBestSell[Flow]
1: INSTALLApplicationServer[Sequence]
   INSTALLDatabaseServer[Sequence]
2: INSTALLOSRedHatLinux73[Invoke]
   INSTALLOSAIX52[Invoke]
3: INSTALLASWAS51[Invoke]
   INSTALLDBMSDB2UDB81[Invoke]
4: INSTALLServletBestsell[Invoke]
   INSTALLTables[Flow]
5: INSTALLOrders[Invoke]
   INSTALLOrderline[Invoke]
   INSTALLItem[Invoke]
   INSTALLAuthors[Invoke]

Figure 4: The BPEL4WS workflow as an IPlanSolution plan structure. Action types are in brackets.

The procedure also computes state-dependency links between the basic activities of the workflow, which become the (executable) actions of the plan as is shown in Figure 5. The links denote that a state is preserved from the producer activity to the consumer activity. If the same state will hold for multiple consumers, as for example in the case of a *flow*, the corresponding links are said to have *equivalent causation*. State-dependency links can be derived directly from BPEL4WS synchronization links, e.g., those that lead to

*Install-ServletBestsell* in the example, or are derived from the semantics of the structural activities. Activities nested within *flow* will have the same starting state, whereas activities nested within *switch* and *pick* have different starting states. The ending state of a preceding basic activity is the starting state of the next basic activity inside a *sequence*. An *init* action that precedes every action and a *goal* action that succeeds every action in the plan are also added. They represent initial and goal states similar to the init and goal plan steps in a partial-order plan.

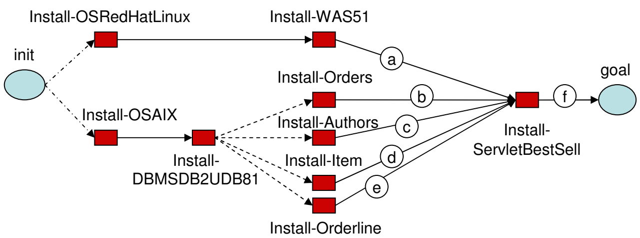


Figure 5: State-dependency links of the example. Two groups of links with equivalent causation occur, illustrated by two types of dashed lines.

If user-supplied action specifications $A$ are available, the pre- and postconditions are directly associated with the actions in the plan. Otherwise, they are derived by *DeriveActivitySpecifications()* explained below.

## Deriving Action Specifications

The analyzer has only access to syntactic structural information plus some limited semantic knowledge of the underlying workflow model. Whereas explicit pre- and postconditions would have provided detailed information about the dependencies of actions in a plan, only very limited dependencies can be derived from the links and the semantics of structured activities. Thus, complete insight into the causalities underlying a plan is impossible. For example, there is no way to discover properties such as that "the same postcondition of a producer is used by multiple consumers". With explicit pre- and postconditions, the producer would have a postcondition that is the precondition of all the consumers. The state-dependency links have unique, but arbitrary names and thus do not allow us to further reason about the states. Consequently, the computed action specifications may contain pre- and postconditions that are neither necessary nor sufficient. The analysis may derive more preconditions than necessary because all conditional dependencies of a *switch* activity are considered as its required preconditions. The postconditions may not be sufficient because there is no guarantee that full causation of an activity is available in the given workflow. However, the pre- and postconditions serve as a good starting point for considering the role of an activity in the workflow and hence for providing metadata.

Figure 6 describes the procedure for basic activities. We interpret state-dependency links as a possible causal dependency on states before and after an action. The preconditions of the earliest action(s) can be interpreted as the initial

*Function*: **DeriveActivitySpecifications**
*Input*: $P$ - Workflow, $L$ - Set of links
*Output*: $A$ - Action Specification with pre- and postconditions

```
For each basic activity n in P
    Create a, a specification corresponding to n
    Populate a as follows:
        For each link in L whose source is n
            Include the link's name in postconditions of a
        End-for
        For each link in L whose target is n
            Include the link's name in preconditions of a
        End-for
    Add a in A
End-for
Return A
```

Figure 6: Derivation of action specifications.

conditions on which the plan is applicable and the postconditions of the latest action(s) are interpreted as the goals that the workflow supports. The following pre- and postconditions are derived for the action *Install-ServletBestSell*:

*Action (Install-ServletBestSell)*
*:parameters ()*
*:precondition: a & b & c & d & e*
*:postcondition: f*

Pre- and postconditions of subplans, which represent those parts of a plan that are nested within a particular structured activity, e.g., the flows or sequences contained in the example, can be derived by common methods of temporal projection (Nebel & Bäckström 1994). In addition, we distinguish *structural* pre- and postconditions, which contain the names of activities that are nested inside a structural activity. This analysis yields more information about what a structured activity actually does. An exception are basic activities of type *receive*, which specify the waiting of the workflow for a message sent by a partner—its name is added to the structural preconditions.

Action (InstallApplicationServer[Sequence])
:parameters ()
:precondition *weakest projected preconditions …*
:structural precondition
:postcondition *strongest projected postconditions …*
:structural postcondition
Install-OSRedHatLinux[Invoke]
Install-WAS51[Invoke]
Install-ServletBestSell[Invoke]

Figure 7: Action specification for activity *Sequence:InstallApplicationServer*.

Figure 7 shows the specification for the sequential activity *InstallApplicationServer*. The structural precondition is empty as there is no nested *receive* activity. The structural postconditions contain the names of the three nested basic activities. The "normal" pre- and postconditions are calculated from the pre- and postconditions of the nested activities—they are omitted from the figure due to space restrictions.

## Deriving Plan Metadata

Given the plan and the (possibly derived) action specifications, various methods are used in *AnalyzePlanInternal()* to analyze the plan. We currently support *structural analysis* of a given plan and a *comparative analysis*, which compares the plan with other plans available from the plan repository. Additional analysis methods, e.g., taking into account execution behavior, can easily be added, but depend on the CHAMPS application.

**Structural Plan Analysis** determines the following properties for a plan:

1. *Length of the plan* as the total number of time steps.
2. *Concurrency of the plan* as the average and maximum number of parallel actions in the time steps.
3. *Exposed postconditions*: postconditions that are not consumed in the next step after being produced are undesirably exposed.
4. The *duration* of a postcondition is the number of time steps between its producer and consumer.
5. A plan is *satisfied* if all of its actions are satisfied. An action is satisfied if all its preconditions are supported by links from some other action.
6. A plan is *justified* if all its actions are remote-linked to the goal action, i.e., there is a path of links between each action and the goal action and no action is redundant.
7. A plan is *safe* if all its links are safe, i.e., not deleted/clobbered by a negative postcondition of another action (threat) between its producer and consumer steps.
8. A plan is *minimally correct* if it is safe, satisfied and justified.
9. A plan has a *critical condition (edge)* if it is only supported by one producer.
10. A plan has a *critical action (node)* if at least one of its preconditions is critical.

Properties 1 and 2 give statistics about the plan. Properties 3 and 4 are very helpful when a plan spans multiple environments, e.g., machines, and when it can have security implications. Properties 5 and 6 follow the terminology and definitions given in (Kim, Gil, & Spraragen 2004), Property 7 is standard from partial-order planning while Property 8 summarizes the collective validity of Properties 5 to 7. We have defined the last two properties to obtain insight into the most crucial elements of the plan. This list is by no means complete, and more properties can be added depending on the application. Additional structural properties, such as finding the most constrained actions having the largest number of pre- and used postconditions, can be easily computed.

For our example plan in Figure 2, the intermediate plan representation illustrated in Figure 5 is generated along with the action specifications. The result of the structural analysis is shown in Figure 8. Further details at individual action or link levels are also provided by the analysis, but not shown here.

Information obtained by the plan analysis can now be used as metadata to manage the plan. Salient features of the processing that can also be used as metadata are:

```
* Analysis performed at Thu Jan 06 17:28:35 IST 2005
_______________________________________________
* Plan analysis completed = true
* Is plan minimally correct ? = true
* Time taken for analysis = 0

DETAILED REPORT BELOW:

* Avg. concurrent actions in plan = 3.0
* Max. concurrent actions in plan = 4.0

* Plan length = 6
* Min. length branch = 4
* Min. sequential plan length = 6
* Max. sequential plan length = 6

* Total conditions not consumed = 0.0
* Total conditions consumed = 9.0
* Total duration of consumed conditions = 13.0
* Detailed consumed & exposed report:
a, duration = 1; b, duration = 1
c, duration = 1; d, duration = 1
...

* Are actions of the plan justified ? = true
* Are links of the plan satisfied ? = true
* Are links of the plan safe ? = true

* Critical actions =
      Install-WAS51, Install-DBMSDB2UDB81, Install-ServletBestSell
      Install-Orders, Install-Orderline, Install-Item, Install-Authors
* Critical predicates =
      links a, b, c, ...
```

Figure 8: Output of structural analysis.

1. Properties of a set of states as metadata. New state-dependency links were created to make the ordering (and thereby causation) among actions explicit. The links represent states between actions and new metadata involving properties of possible states can be recorded, e.g., equivalent causation.
2. The specifications of actions and their properties as metadata. Actions *Install-Orders*, *Install-Authors*, *Install-Item*, and *Install-Orderline* are executed in the same context (pre- and postconditions). Hence, they may be performing similar things and, if additional metadata suggests, they may be interchangable.
3. New statistics from analysis output as metadata. *Install-ServletBestSell* is the most constraining action. This is deducible by a simple count of the number of pre- and postconditions of an action.

As noted in the introduction, the benefit of our approach is that the metadata is automatically produced and hence, uniform and reproducible, as the plan may get altered over time.

**Comparative Plan Analysis** Plan Analysis can also be performed using a repository of previously stored plans, which may optionally have been annotated with metadata from previous analyses. Some of the interesting analyses in the presence of other plans are:

1. Plans that achieve similar goals.
2. Plans that share actions.
3. Plans that were derived from workflows of similar structure.
4. Plans that share metadata.

Similarity information is either explicitly available from some metadata annotations, where it was perhaps provided by a human, or it must be computable from the information contained in the plan, its action specifications, and the semantics of the underlying workflow language. In our application, we retrieve similar plans by querying the plan repository, which returns all plans matching the query. We describe the repository architecture and its querying process in more detail below.

## Plan Repository

For AI planning research, plans are isolated objects. In planning applications, plans must usually be associated with additional information in order to make them usable in the real world. For example, our CM workflows represented in BPEL4WS must be associated with web service specifications in the WSDL standard (Christensen & others 2002), XML schema, and metadata that determines when a plan is considered for execution or when its life cycle expires. The plan repository supports such associations and allows plans to be grouped in folder-like structures. Figure 9 shows the association of BPEL4WS documents with documents of other types and the maintenance of the association links between these documents in a descriptor document.

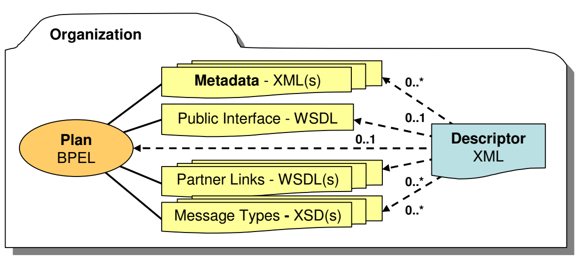


Figure 9: Putting plans into context by associating information in the repository.

Figure 10 shows the architecture of the repository (Vanhatalo 2004). The query engine and data persistence mechanism are pluggable components. In the current prototype implementation, all information is persisted in a file system, but a database will be used in the future. The CHAMPS application, Planner4J, and the Plan Analysis component manipulate the plans as Java objects. The repository provides the serialization mechanism for these objects using the Eclipse Modeling Framework (Eclipse-Org. 2004) with the advantage that any application can work with the information stored in the repository without being aware of its internal serialization which is hidden by the repository API.

As a novel feature, the repository API provides an object-oriented query mechanism, which allows applications to query items in the repository without knowing how these items are physically stored. The current implementation uses query engines based on the Object Constraint Language (OCL) (Warmer & Kleppe 2003). This design frees developers from the XML burden and allows them to concentrate on the object model of the application, which they usually know very well. A major advantage of OCL over XQuery is its

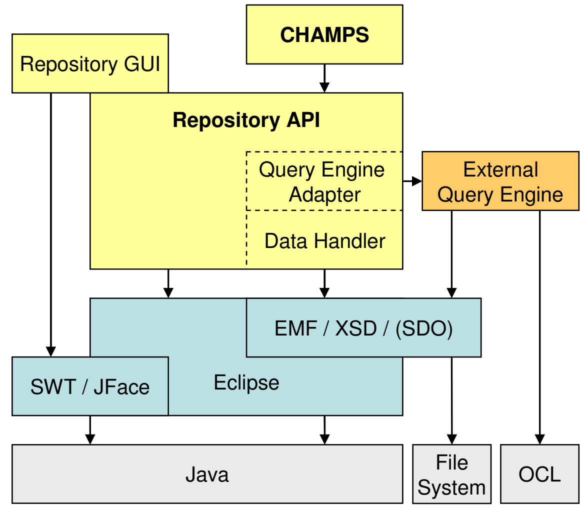


Figure 10: Repository architecture showing components and the technology used.

ability to navigate in the data model following all the associations in an object model, while XQuery forces any application to formulate its query based on the tree structure of the underlying serialization-specific XML schema. The repository works with two OCL engines–an open-source OCL query engine from the University of Kent (Akehurst & Patrascoui 2004) and an IBM-internal OCL query engine.

Using the Kent OCL engine, the querying performance of the repository was tested on a sample collection of 165 BPEL4WS workflows associated with 1056 other XML documents occupying approx. 5 MB on the hard disk. On an IBM ThinkPad A31p laptop with 1 GB of main memory and an Intel Pentium 4 (1.70GHz) processor, 165 workflows can be queried within 2.8 sec. The time needed to query the files increases linearly with the number of files. This is a satisfactory performance for our prototype when taking into account that so far no database or indexing mechanisms are used.

## Using Metadata to Manage the Plan Life Cycle

The repository groups each plan (in our case a BPEL4WS workflow) and its metadata into a folder-like structure called an *organization*. The organizations form a tree structure similar to a directory structure of a file system. This enables the storage of related plans under the same organization subtree, and speeds up queries by focusing on particular subsets of organizations.

Often a plan is searched from the repository based on the metadata that is stored in a separate document. The query mechanism takes this into account, thus enabling a query into the metadata, but returns a reference to the related plan instead of the reference to the metadata satisfying the query. Instead of references, the EMF objects, e.g., the entire BPEL4WS plan, can directly be returned for further processing, e.g., plan execution.

As an example query, we search for all plans that contain the *Install-ServletBestSell* activity and that are minimally correct. To answer this query, plan as well as metadata documents must be queried. First, the plans containing the *Install-ServletBestSell* activity are found by executing the following OCL query over all the plans in the repository:

```
context process::TProcess inv:
TActivity.allInstances()->exists(a |
a.name='Install-ServletBestSell')
```

The query parameters are set such that for each matching plan a reference to its metadata is returned as the query result. Second, we query this subset of metadata with an OCL query that finds all the minimally correct plans:

```
context planner::PlannerMetaData
inv: self.minimallyCorrect
```

The query parameters are now set such that the plans that are associated with the matching metadata document are returned. This flexibility of the query mechanism allows applications to easily inspect, retrieve, replace plans stored in the repository, and to reorganize the repository organizations.

## Conclusion and Future Work

In this paper, we identify open AI research problems by discussing an autonomic computing application that requires solutions for plan analysis and plan life-cycle management. We demonstrate the need to look at plans beyond their synthesis and to develop techniques that help to manage the life cycle of plans within an application and based on the context of their usage. We discuss that plans must be analyzed that have no explicit action model, initial state and goals, and which may be the outcome of an unknown planning system (be it human or domain-dependent)—which is in major contrast to current AI planning solutions. We investigate and extend plan analysis techniques that automatically generate metadata annotations of plans. To the best of our knowledge, metadata annotations to drive the life cycle of plans have not been considered so far.

In particular, we consider more expressive representations of plans in the form of workflows, which contain nested activities and expressive control structures. The analysis involves evaluating a plan in isolation and by comparison with a collection of previous plans. A plan repository is presented that allows an application to organize large sets of plans and that provides expressive query and retrieval mechanisms, which can exploit the metadata associated with a plan. The interaction of plan analysis and plan life-cycle management using the repository is discussed within an autonomic computing application.

Future work will concentrate on increasing the range of analyses and improved tool support. We are exploring techniques to provide insight into the execution aspects of plans. Moreover, the development of an integrated environment for analyzing plans, selecting metadata, and querying plans from a repository will be pursued further.

## References


Akehurst, D., and Patrascoui, O. 2004. Object constraint language library. Technical report, http://www.cs.kent.ac.uk/projects/ocl/index.html.

Christensen, E., et al. 2002. The web services description language WSDL. http://www.w3.org/2002/ws.

Curbera, F., et al. 2002. Business process execution language for web services (bpel4ws). http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.

Eclipse-Org. 2004. Eclipse modeling framework. http://www.eclipse.org/emf/.

Fu, X.; Bultan, T.; and Su, J. 2004. Analysis of interacting bpel web services. In *13th International World Wide Web Conference (WWW'04)*. ACM Press.

Garland, A., and Lesh, N. 2002. Plan evaluation with incomplete action descriptions. In *18th National Conference on Artificial intelligence*, 461–467. AAAI Press.

Kambhampati, S. 1990. A classification of plan modification strategies based on coverage and information requirements. In *AAAI 1990 Spring Symposium on Case Based Reasoning*.

Keller, A.; Hellerstein, J.; Wolf, J.; Wu, K.; and Krishnan, V. 2004. The CHAMPS system: Change management with planning and scheduling. In *Proceedings of the IEEE/IFIP Network Operations and Management Symposium*.

Kim, J.; Gil, Y.; and Spraragen, M. 2004. A knowledge-based approach to interactive workflow composition. In *ICAPS-04 Workshop on Planning and Scheduling for Grid and Web Services*.

Myers, K. 2004. Toward a theory of qualitative reasoning about plans. In *ICAPS-04 Workshop on Connecting Planning Theory with Practice*.

Nebel, B., and Bäckström, C. 1994. On the computational complexity of temporal projection. *Artificial Intelligence* 66(1):125–160.

Pollack, M. E. 1992. The uses of plans. *Artificial Intelligence* 57(1):43–68.

Srivastava, B.; Bigus, J.; and Schlosnagle, D. 2004. Bringing planning to autonomic applications with able. In *1st Intl. Conf. on Autonomic Computing (ICAC'04)*, 154–161.

Srivastava, B. 2004. A software framework for applying planning techniques. In *Proc. Knowledge Based Computer Systems, Hyderabad*. Also as IBM Res. Report RI04001.

TPP-Council. 2002. Transaction processing performance council benchmark W specification (web commerce) v1.8. http//www.tpc.org/tpcw.

Vanhatalo, J. 2004. Building and querying a repository of BPEL process specifications. Master's thesis, Helsinki University of Technology, Finland.

Warmer, J., and Kleppe, A. 2003. *The Object Constraint Language - Getting Your Models Ready for MDA*. Pearson Education, Inc., MA, USA.

Yaman, F.; Adali, S.; Nau, D.; Sapino, M.; and Subramanian, V. 2004. Plan databases: Model and algebra. In *Proc. International Conference on Foundations of Information and Knowledge Systems (FoIKS 2004)*.