RI 02025 15 January 2003 Computer Science

IBM Research Report

PlanSP: A Framework to Automatically Analyze Software Development and Maintenance Choices

Biplav Srivastava
IBM Research Division
IBM India Research Lab
Block I, I.I.T. Campus, Hauz Khas
New Delhi - 110016. India.

PlanSP: A Framework to Automatically Analyze Software Development and Maintenance Choices

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

Abstract

In software engineering, a piece of software is assembled from components or modules and these components in turn can be recursively made up from smaller sub-components. The management of a software project involves tracking the development and maintenance of the individual components. However, this brings to fore the crucial issue of how to manage components in such a way that they could be leveraged effectively – reusing existing components and monitoring their evolution during the life cycle of the software. Currently, when a new piece of software is being created, the user manually has to evaluate the relevance of existing components based on her development objectives like time available and expected performance metric. Similarly, when a software has been released and is now being maintained, any updates to the dependent components is evaluated, either manually or blindly (through timestamps), to decide if a new build of the software is necessary. Though tools exist to track component dependencies and historical changes, the key software management hurdle is the manual evaluation of the trade-offs. With the growing trend in software engineering to build and distribute software components as web services that can be invoked across platforms and languages in a distributed environment, there is a real need of automated solutions to guide the developer in building and maintaining complex applications.

We introduce an automated decision-support framework for software development and maintenance called PlanSP that can analyze different choices and assist the user in making cost-effective decisions. Our approach is to build a formal model of the software and use automated planning/ reasoning techniques to produce alternative choices ("plans") to develop or maintain the software under consideration while respecting the user's effort and performance objectives. We have built a proof-of-concept prototype to demonstrate that the PlanSP framework is both useful and practical for software project management.

1 Introduction

As software systems[Pressman1996] become complex, organizations have to put significant investment in their development and maintenance (i.e., software life cycle). Any piece of software is usually organized as a collection of components or modules. A key challenge in management of a software life cycle is how to effectively leverage existing software components while injecting resources to: (a) develop and support software that perform to required specification and (b) deliver the software in a timely manner.

Consider a typical software project management scenario. Humans devise the Work Breakdown Structure (WBS)[Moder & Phillips1964] to identify the different tasks at some granularity and input this information to a project management tool along with estimates on time and resources for each task. Microsoft Project[Microsoft1998] is a standard tool used in industry for scheduling activities. It has elaborate guidelines on how to reason about a project - find the critical path in the project, compute slack time for individual tasks, evaluate tasks to identify over-allocated resources, etc. It is not hard to see that the user, who may be a project manager or software architect, has to evaluate the relevance of existing components *manually* based on the project objectives like expected software functionality, performance and development time. This analysis also helps the user scope out new development in the project and estimate the overall integration effort involved.

Now consider the case when a software has been released and is now being maintained. If any updates/patches are available for the software components that were reused in the project, their impact is *manually* evaluated to decide if a new build of the software is necessitated. Though there are some tools to track component dependencies and record history of component releases, the key software management hurdle still remains that the trade-off choices have to be manually evaluated.

Our contribution relates to this general area of software project management. We propose a framework to analyze different project management choices *automatically* in the following reasoning problems and thereby assist the user make informed decisions.

- Scenario 1: When creating a new piece of software S_{new}, help the user:

  [Problem 1] `Reuse`: Find existing components B_i that can accomplish part of the functionality needed in S_{new}. Hence, find components that are candidates for reuse in the project.

  [Problem 2] `Reduce`: Identify existing components B_i that reduce scope and complexity of any new development on S_{new} without sacrificing on S_{new}'s functionality.

- Scenario 2: While an existing software S_{new} is being maintained, help the user:

  [Problem 3] `Aware`: Identify (sub-)components B_i whose newly released enhancements can affect the functionality of S_{new}.

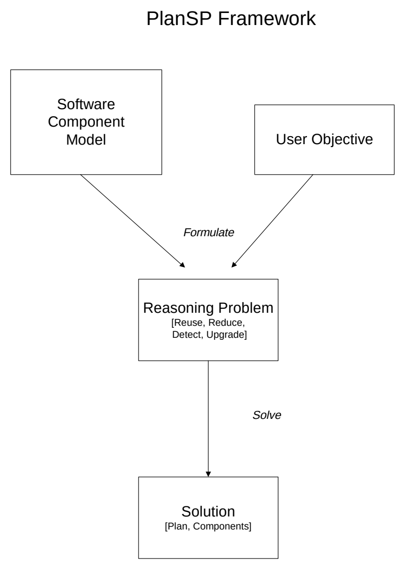


Figure 1: *PlanSP approach of decision support in software project management.*

> [Problem 4] `Upgrade`: Evaluate and incorporate new enhancements of (sub-)components B_i that are necessary for enhancing the objective of S_{new}.

Our solution approach is to build a formal model about the capabilities of software components and consider listed project management problems as shown in Figure 1. The model is based both on the structured information that is available about a software at its release time (like dependency information used by $Make$) and optionally, measurements such as performance/ cost metric and expected integration effort (time). The latter will be used to automatically reason with usage tradeoffs and hence, it makes sense for the user to leverage any historical information gained in this regard. The formal model for software components is stored and appropriately referenced by PlanSP.

Given the software model of components and user objective, the decision-support issues are cast into reasoning problems and solved. We use automated AI planning [Weld1999]/ reasoning techniques to produce alternative choices ("plans") to develop or maintain the software under consideration. *The novelty of the paper is in how the problems are created for tackling the specific decision-support issues and the adaptation of planning algorithms to solve them.* The goal is not only to support the functionality of the software but also to respect the user's effort and performance objectives. We demonstrate that the PlanSP framework is both useful and practical for software project management.

There has not been much use of automated reasoning techniques in software reuse and maintenance. Previously, [Huff & Lesser1988] had proposed using planning techniques to automate the software development process. However, their main focus was on automating the compile/ build, test and release cycles rather than software reuse. A limited search-based component retrieval solution has been proposed by Hall[Hall1993] but the search space is restricted to depth 3. More related work is discussed at the end.

The paper is arranged as follows: we start with illustrative examples of how an automated tool can help reason with choices over the course of software life cycle and assist a user in project management. We then describe the details of how a formal model of the software is built and describe the automated reasoning/ planning techniques. Next, we discuss the inputs and assumptions of our framework and present algorithms for solving the decision support problems. We demonstrate that PlanSP is reasonable from a practical standpoint with examples increasing in domain expressivity. We conclude with a discussion of related work, contributions and future challenges.

2 Some Example Scenarios

Consider a fictitious software company named `WordSoft` that builds multi-lingual text analysis applications. Now a developer at `WordSoft`, whom we will call Jack, wants to avail of decision support for project management.

2.1 Example: Module Selection for Multi-Lingual Text Analysis

In text domain, `WordSoft`, has built 4 types of language specific reusable components in a library: *tokenizer*, which identifies tokens in a text document, *lemmatizer/stemmer*, which given tokens, can find the lemma and inflection variants, *synonym-lookup*, which finds synonyms of a lemma and *thesaurus-lookup*, which find thesaurus details for a lemma. Suppose the current set of supported languages are English, German, Spanish and Hindi, making a total of 16 components in the reuse library. Figure 2 shows the domain pictorially. Information about a component is represented as ground predicates and the convention is that dependencies (preconditions or inputs) are shown at the lower left side of a box while the functionalities (effects or outputs) are shown at the upper right corner.

Now Jack has to implement the following functionality in a new software S_{new}: given a multi-lingual document set, find lemmas in Spanish and thesaurus entries in German. Jack wants to determine if there are components in the library that can collectively fulfill the requirement. If this is not possible, he is interested in leveraging the exisiting components to reduce any new development for S_{new}. Figure 3 shows how PlanSP can automatically meet Jack's requirement.

Now consider the case when some component in the library (e.g., *tokenizer* for English) gets modified and released. Jack would want to know if there is an

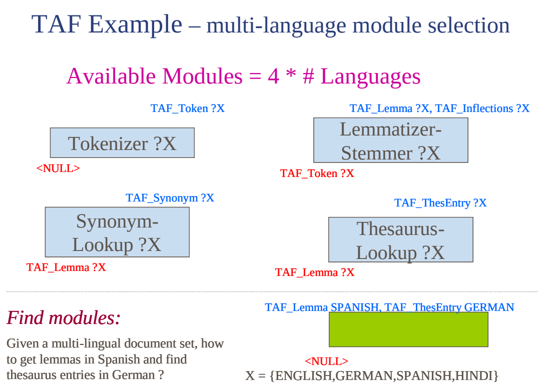

TAF Example – multi-language module selection
Available Modules = 4 * # Languages
TAF_Token ?X
Tokenizer ?X
<NULL>
TAF_Lemma ?X, TAF_Inflections ?X
Lemmatizer-Stemmer ?X
TAF_Token ?X
TAF_Synonym ?X
Synonym-Lookup ?X
TAF_Lemma ?X
TAF_ThesEntry ?X
Thesaurus-Lookup ?X
TAF_Lemma ?X
Find modules:
Given a multi-lingual document set, how to get lemmas in Spanish and find thesaurus entries in German ?
TAF_Lemma SPANISH, TAF_ThesEntry GERMAN
<NULL>
X = {ENGLISH,GERMAN,SPANISH,HINDI}

Figure 2: *The domain on multi-lingual module selection.*

existing software S_{old} which is being maintained and is likely to be impacted. If S_{old} was the S_{new} of above, no update is needed since no English tokenizer is part of the released software.

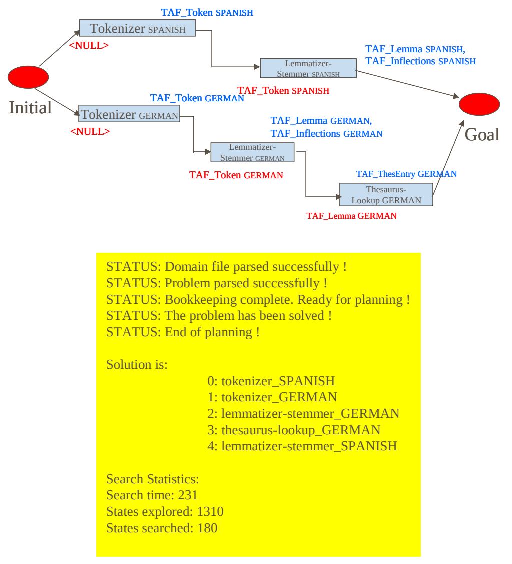
2.2 Example: Cost-Effort Reasoning with Text Mining Components

Let us now focus on some software components that `WordSoft` has built for Text Mining (TM) in Figure 4. The available software components are a clustering algorithm, *Clusterer-1*, with two reusable sub-components, C_1^1 that takes numeric input and produces an hierarchical cluster, and C_2^1, which takes a cluster hierarchy and displays it; *Wordconv-1*, a converter for words to numeric form (W_1^1) and a text miner, *TextMP-1* with two sub-components T_1^1, which takes text input and returns patitioned clusters and T_2^1, which displays partitions. The performance and effort estimates of each component is shown in brackets, respectively.

Jack's objective is to build a new text miner software that can take text as input and shows a hierarchical view of the clusters with the constraint that its performance be within 5 seconds and not take more than 10 hours to build. He will like to know the various options for component reuse, and also figure out the scope of any new development.

3 Towards a Formal Model for Reasoning

We begin by discussing the foundational issues in representation of software components and automated reasoning. In the next section, we use this background while solving the 4 decision-support problems.

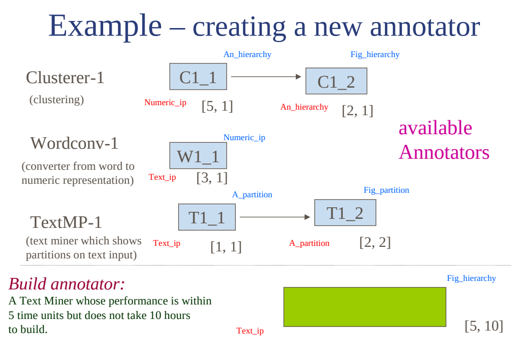


```
STATUS: Domain file parsed successfully !
STATUS: Problem parsed successfully !
STATUS: Bookkeeping complete. Ready for planning !
STATUS: The problem has been solved !
STATUS: End of planning !

Solution is:
                    0: tokenizer_SPANISH
                    1: tokenizer_GERMAN
                    2: lemmatizer-stemmer_GERMAN
                    3: thesaurus-lookup_GERMAN
                    4: lemmatizer-stemmer_SPANISH

Search Statistics:
Search time: 231
States explored: 1310
States searched: 180
```

Figure 3: *Finding the relevant components from the library.*


Example – creating a new annotator

Build annotator:
A Text Miner whose performance is within 5 time units but does not take 10 hours to build.

Figure 4: *A domain for software components with cost and effort information.*

```
all: c1 c2 link

c1:
    gcc -c foo.c

c2:
    gcc -c moo.c

link:
    gcc foo.o moo.o -o software
```

Figure 5: *Dependencies specified in a Makefile.*

3.1 Representation of Software Components

We characterize a piece of software by the functionalities it provides, the other software components that it depends on, a measure to gauge its performance (like runtime) and the amount of effort that is needed to build it. We elaborate on how these inputs may be obtained.

3.1.1 Functionality

The functionality about a piece of software component, i.e. its effects, is represented by predicates. Predicate, a construct from logic, refers to relationship among objects where the latter are represented by terms (constants or variables). If any term in a predicate is a variable, the predicate is also called a parameterized or lifted predicate, else it is a ground predicate. Example of a parameterized predicate is (Document ?x) and its instantiation, (Document ENG), is a ground predicate refering to a document in English. Semantics, or the meaning of a predicate, is both assigned by, and meant for the humans/ applications in the domain.

Since functionality about a piece of software is known to the software creators, often as part of their requirement gathering step, one way to obtain it can be from the software developers as part of the software release information. Another source is the user groups which collect data about practical usage of components.

3.1.2 Dependency

We represent dependencies of a piece of software component, i.e. its precondition, by predicates. An important source of this information is the *Make* files used to build the software. This is illustrated in the simple example of Figure 5. The example shows that to build this software ("all"), the build dependency is on C1, C2 and link steps. C1 and C2 are potential software dependencies.

Eventually, the precise software dependencies is known completely only to the software creators. Hence, another way to obtain it can be by requiring it from them as part of the software release information.

3.1.3 Performance or Cost

The performance measure of a software depends on a variety of factors ranging from the context in which the software is used to the time it takes to complete its activity. One can generalize this measure as some numeric cost of executing the software where the cost information is of importance to the decision maker.

One simple cost measure can just be the estimated execution time on a typical data size like "1sec for XML parser on 1MB XML data". Such data can be obtained from performance testing or field trials. Another example is the precision information about data mining components on specific types of inputs - "label persons in a document with 0.9 accuracy".

3.1.4 Effort

The effort required in integration of a software component with other software pieces depends on a variety of factors like the software development methodology, the organization of the software reuse library, the skill of the developers, the details of the components, etc. However, over time, effort information as a temporal quantity (e.g., 1 week) can be collected from actual usage of the components and provided to solve the software decision problems. Another source for obtaining this information are the experienced software developers who have used similar components in their projects before.

3.1.5 Example

Figure 6 shows the representation of domain for the module selection example of Section 2.1. The representation is expressed in Planning Domain Description Language (PDDL[Fox & Long2002]) with STRIPS[Fikes & Nilsson1990] semantics. This means that software components are modeled as deterministic actions with preconditions that must hold for the actions to be applicable, and instantaneous effects. Moreover, there are no additional actors in the modeled domain (world).

3.2 Automated Reasoning

A planning problem[Weld1999] P is a 3-tuple $\prec I, G, A \succ$ where I is the complete description of the initial state, G is the partial description of the goal state, and A is the set of executable (primitive) actions. A state T is simply a collection of facts with the semantics that information corresponding to the predicates in the state holds (is true). An action A_i is applicable in a state T if its precondition is satisfied in T and the resulting state $T^{'}$ is obtained by incorporating the effects of A_i. An action sequence S (a plan) is a solution to P if S can be executed from I and the resulting state of the world contains G. A type of planners, called state-space planners because they search in the space of possible world state, is shown in Figure 7. As shown, `FindSequence` can handle problems where information represented as predicates. We will use it as the base planner to illustrate the various algorithms but the paper is also

```
;; An text domain of TAF
(define (domain text)
    (:requirements :strips)

;; Types hierachy
(:types object)

;; Predicate list
(:predicates (TAF_Token ?x) (TAF_Lemma ?x)
       (TAF_Inflections ?x)(TAF_Synonym ?x)
       (TAF_ThesEntry ?x))

;; Action list
(:action TOKENIZER
      :parameters (?x)
      :precondition
      :effect (TAF_Token ?x))

(:action LEMMATIZER-STEMMER
      :parameters (?x)
      :precondition (TAF_Token ?x)
      :effect (and (TAF_Lemma ?x) (TAF_Inflections ?x)))

(:action SYNONYM-LOOKUP
      :parameters (?x)
      :precondition (TAF_Lemma ?x)
      :effect (TAF_Synonym ?x))

(:action THESAURUS-LOOKUP
      :parameters (?x)
      :precondition (TAF_Lemma ?x)
      :effect (TAF_ThesEntry ?x)))
```

Figure 6: *The specification of components for multi-module selection in PDDL.*

applicable to the other types of planners called the plan-space planners (i.e., planners which reason in the space of plans – partial solutions).

The insight in posing a software project as a planning problem is that the initial state I is nothing but input data specification while goal state G is the functionality desired from the software. Each component present in the software library is an action A_i with its inputs being the preconditions and its outputs being the effects.

Now any suitable planner that can handle the particular information representation can be used to synthesize the sequence of actions to achieve the goal. The possibilities are:

- If only functionality and dependency information is available, a planner like `FindSequence` can be used. We will use our Java implementation called ParamC which takes input in the STRIPS[Fikes & Nilsson1990] notation (predicates) and uses goal distance heuristics to guide its search. It can build the solution plan either in the forward or backward direction[1]. However, we could have used any planner that supports STRIPS like Graphplan[Blum & Furst1995] for equivalent results.

- If performance and time estimates are additionally available about components, a cost-metric planner can be used to select among potential plans to meet some optimization criteria. Example: How to build a software such that its performance is within 5 seconds but it does not take 10 hours to build ? We will consider this situation in Section 5.

[1]All results use forward direction.

```
FindSequence(I, G, A)
     1. If G $\subset$ I
     2.    Return
     3. End-if
     4. Ninit.sequence = {}; Ninit.state = I
     5. Q = Ninit
     6. While Q is not empty
     7.    N = Remove an element from Q (*heuristic choice*)
     8.    Let S = N.sequence; T = N.state
     9.    For each component $A_i$ in A
     10.       If precondition of $A_i$ is satisfied in state S
     11.          Create new node $N'$ with:
                  $N'$.state = Update(S, effects of $A_i$) and
                  $N'$.sequence = Append(N.sequence, $A_i$)
     12.       End-if
     13.       If $N'$.state $\supset$ G
     14.          Return $N'$ // *Return a plan*
     15.       End-if
     16.       Q = Q $\cup$ $N'$
     17.    End-for
     18. End-while
     19. Return FAIL // *No plan was found*
```

Figure 7: *Template of a standard state-space planner that can reason with information of components (actions) represented as predicates.*

Insight from incomplete plans: If no complete sequence of components (plan) is possible from the library for a given requirement, planning can still help the user scope down the requirement of any new development that must be done to attain all the functionalities. To get estimate of new development, the planner has to sort the search space of non-solutions based on heuristic distance to goal. The plan with the lowest such heuristic gives us the candidate plan requiring new development.

4 Solving Project Management Decision-Support Problems

We elaborate how decision-support problems that arise during software life cycle are managed in PlanSP. We will use the module selection example (see Section 2.1) in this section.

4.1 Scenario 1 - Software Development

Suppose the user wants to build a new software S_{new} and has a software components library.

4.1.1 Reuse

In this case, the user wants to evaluate the existing components B_i that can accomplish part of functionality of S_{new}. Hence, he wants to find components that are candidates for reuse from the library. Figure 8 shows the formulation of the solution and Figure 9 presents an algorithm to find relevant components. `FindRelevantComponent` determines relevant components by starting from the

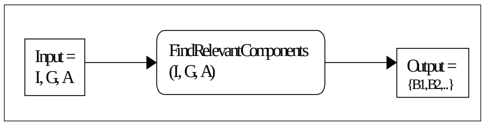


Figure 8: *The solution to* `Reuse` *problem.*

```
FindRelevantComponent(I, G, A)
    1. RelevantComps = {}
    2. If G $\subset$ I
    3.    Return RelevantComps
    4. End-if
    5. T = G; Tlast = {}
    6. While T not equal to Tlast // Loop while T grows
    7.       Select a component $A_i$ in $A$
    8.       If effect of $A_i$ is satisfied in state T
    9.          RelevantComps = RelevantComps $\cup$ $A_i$
    10.         Tlast = T
    11.         T = T $\cup$ $A$.precondition // Update T
    12.      End-if
    13. End-while
    14. Return RelevantComps
```

Figure 9: *An algorithm to find related components for achieving G.*

goal specification and finds components that possibly support the goals or iteratively other components that in turn supports the goal.

Figure 10 shows the planning specification of the module selection problem of Section 2.1 and the solution to the `Reuse` using `FindRelevantComponent` algorithm. It lists the specific modules and their languages in the library which are relevant to the given problem. No other modules are provably relevant other than the 5 listed for this example.

4.1.2 Reduce

In this case, the user wants to evaluate the existing components B_i that can accomplish maximum functionality of S_{new}. However, she wants to optimize development by finding how to use relevant B_i to minimize scope and complexity of the new software development for S_{new}. Figure 11 shows the generalized formulation of the `Reduce` solution (optional formulation in the presence of cost and effort metric is in brackets) and Figure 12 gives the algorithms to solve the simplest case (without any cost and effort information). In algorithm `FindEffectiveSequence`, first an attempt is made to produce a plan which

```
;; Find the multi-lingual modules
(define(problem simple)
        (:domain text)
        (:objects ENGLISH GERMAN
                  SPANISH HINDI)
        (:init)
        (:goal (and (TAF_Lemma SPANISH)
                (TAF_ThesEntry GERMAN))))
```

```
STATUS: Domain file parsed successfully !
STATUS: Problem parsed successfully !
** Relevant components *** :
       [lemmatizer-stemmer_SPANISH,
       lemmatizer-stemmer_GERMAN,
       tokenizer_SPANISH,
       tokenizer_GERMAN,
       thesaurus-lookup_GERMAN]
```

Figure 10: *Planning problem for module selection example and the result for* `Reuse` *scenario.*

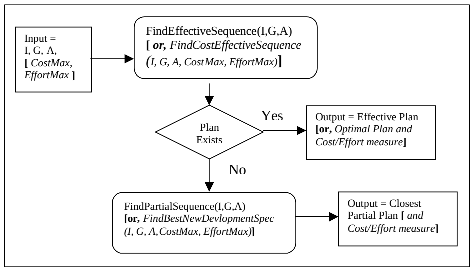

Input = I, G, A, [CostMax, EffortMax]
FindEffectiveSequence(I,G,A) [or, FindCostEffectiveSequence (I, G, A, CostMax, EffortMax)]
Plan Exists
Yes
Output = Effective Plan [or, Optimal Plan and Cost/Effort measure]
No
FindPartialSequence(I,G,A) [or, FindBestNewDevlopmentSpec (I, G, A,CostMax, EffortMax)]
Output = Closest Partial Plan [and Cost/Effort measure]

Figure 11: *The solution to* `Reduce` *problem.*

can meet the desired functionality from the library components[2]. If no plan is possible, the partial plan leading to the closest state to the goal is returned as this corresponds to the minimum development overhead (with respect to the distance metric).

For the multi-module problem, we use the goal metric that minimum number of components (actions) should be utilized. Figure 3 shows plan returned by `FindEffectiveSequences`. It gives the actual plan of how the tokenizer and lemmatizer in Spanish, and the tokenizer, lemmatizer and thesaurus module in German can achieve the objective. Though it is not evident in this example, it is very likely that the number of components in the plan shown by `Reduce` is different (and smaller) from that returned by `Reuse` since the latter only gives a set of relevant components while `Reduce` shows a definite ordering among a subset of them.

In the general case, we envisage that the developers can optionally give additional information about the performance of the component and expected integration effort (time) with it. Given this information, we present algorithms in Section 5 about how automatic reasoning about performance v/s effort tradeoff

[2]By default, a heuristic planner searches states in the order of their heuristic value and any metric/heursitic function can be supplied.

```
Let Metric denote a state distance measuring function.

FindEffectiveSequence(I, G, A)
    1. If G ⊂ I
    2.     Return {}
    3. End-if
    4. Plan = FindSequence(I, G, A)
    5. If Plan != FAIL
    6.     Return Plan
    7. Else
    8.     Return FindPartialSequence(I, G, A)
    9. End-if


FindPartialSequence(I, G, A)
    1. Call FindSequence(I, G, A) but return partial plan
       in the search queue with lowest distance to the goal,
       as measured by the default goal metric.
```

Figure 12: *An algorithm to solve* `Reduce` *problem.*

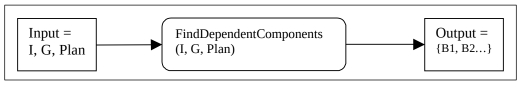


Figure 13: *The* `Aware` *problem.*

can be performed.

4.2 Scenario 2 - Software Maintenance

This scenario occurs when a software S_{old} has been released and is now being maintained.

4.2.1 Aware

In the `Aware` case, the user is given the formal specification of software S_{old} and the plan used to create it. The user wants to detect sub-components B_i whose enhancements can be of possible interest while maintaining S_{old}. Figure 13 shows the formulation of the solution and an algorithm is shown in Figure 14 to find the dependent components. Dependent components are nothing but the components that come together to make up the functionality of a software.

In the module selection example, the dependent module are shown in Figure 15. Hence, they are the component that the application has to be aware of for releases.

```
FindDependentComponents(I, G, Plan)
    1. DependentComps = {}
    2. For each component $C_i$ in Plan
    3.     DependentComps = DependentComps $\cup$ $C_i$
    4. End-for
    5. Return DependentComps
```

Figure 14: *An algorithm to solve the* `Aware` *problem.*

```
STATUS: Domain file parsed successfully !
STATUS: Problem parsed successfully !
** Dependent components *** :
        [tokenizer_SPANISH,
        tokenizer_GERMAN,
        lemmatizer-stemmer_GERMAN,
        thesaurus-lookup_GERMAN,
        lemmatizer-stemmer_SPANISH]
```

Figure 15: *Dependent components for the module selection problem.*

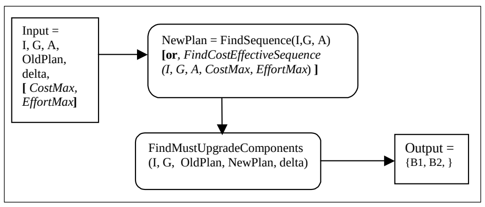

Input = I, G, A, OldPlan, delta, [CostMax, EffortMax]
NewPlan = FindSequence(I,G, A) [or, FindCostEffectiveSequence (I, G, A, CostMax, EffortMax)]
FindMustUpgradeComponents (I, G, OldPlan, NewPlan, delta)
Output = {B1, B2, }

Figure 16: *The solution to the* `Upgrade` *problem.*

```
FindMustUpgradeComponents(I, G, OldPlan, NewPlan, delta)
    1. UpgradeComps = {}
    2. If G $\subset$ I or OldPlan = NewPlan or
         DifferenceMetric(NewPlan, OldPlan) $\prec$ delta
    3.       Return UpgradeComps
    4. End-if
    5. For each component $C_i$ in NewPlan
    6.     If $C_i$ is not present in OldPlan or
           DifferenceMetric($C_i$, OldPlan.$C_i$) $\succ$ delta
    7.         UpgradeComps = UpgradeComps $\cup$ $C_i$
    8.     End-if
    9. End-for
    10. Return UpgradeComps
```

Figure 17: *An algorithm to solve the* `Upgrade` *problem.*

4.2.2 Upgrade

In this case, the user is given the formal specification of software S_{old} and the plan used to create it. The user wants to detect sub-components B_i whose enhancements must be incorporated while maintaining S_{old} because it affects the functionality of S_{old}. Note that the action (component) library A used to create S_{old} may have changed now.

Figure 16 shows the formulation of the solution and Figure 17 gives an algorithm to find components that necessarily must be upgraded. The algorithm relies on the `DifferenceMetric` function to compare software components in plans based on their capabilities. The plans can be compared on the same metric that was used to create them in the first place (e.g., plan length, cost and effort information, precision information, etc.).

5 Decision Support with Cost / Performance and Effort Measures

The model of the software component can be generalized to include a measure to gauge the component's performance (like precision or runtime) and the amount of effort needed to integrate it. Refering to the text mining example in Section 2.2, Jack (the user) wants to build a new text miner software and has requirements for both the functionality (S_{new} will take text as input and show hierarchical view of the clusters), and constraints (it should perfom within 5 time units and take less than 10 hours to build).

The formulation of the `Reduce` problem with cost and performance information is also shown in Figure 11[3]. Figure 18 and Figure 19 show algorithms

[3]Another formulation could be that the user is interested in a set of plans within a [CostMin, CostMax] and [EffortMin, EffortMax] range. Now, both existing possibilities and partial plans

```
FindCostEffectiveSequence(I, G, A, CostMax, EffortMax)
     1. If G ⊂ I
     2.    Return {}
     3. End-if
     4. MetricMax = CalculateMetric(CostMax, EffortMax)
     5. Ninit.sequence = {}; Ninit.state = I; Ninit.cost = 0; Ninit.effort = 0
     6. Q = Ninit
     7. While Q is not empty
     8.       N = Remove an element from Q
     9.       Let S = N.sequence; T = N.state
     10.      For each component $A_i$ in $A$
     11.            If precondition of $A_i$ is satisfied in state S and
                    CalculateMetric(Ninit.cost, Ninit.effort) $\prec$ MetricMax
     12.               Create new node $N^{'}$ with:
                       $N^{'}$.state = Update S with result of effect of $A_i$ and
                       $N^{'}$.sequence = Append(N.sequence, $A_i$)
                       $N^{'}$.cost = update(N.cost, $A_i$.cost)
                       $N^{'}$.effort = update(N.effort, $A_i$.effort)
     13.            End-if
     14.            If G $\subset$ $N^{'}$.state and
                    CalculateMetric(Ninit.cost, Ninit.effort) $\prec$ MetricMax
     15.               Return $N^{'}$ // Return a plan
     16.            End-if
     17.            Q = Q $\cup$ $N^{'}$
     18.      End-for
     19. End-while
     20. Return FAIL // No plan was found


CalculateMetric(cost, effort)
     1. Return evaluation on the desired function combining
        cost and effort measures
```

Figure 18: *One formulation of the reduce problem where a single plan is output.*

FindCostEffectiveSequence and FindBestNewDevlopmentSpec to find existing and partial plans, respectively, with the new information. They use routine CalculateMetric to derive a metric from the performance and effort estimates, and use this metric to guide search. In FindCostEffectiveSequence, the search progresses in a branch-and-bound manner over the space of possible action sequences. Hence, it is guaranteed to give optimal plan with respect to the input metric. In FindBestNewDevlopmentSpec, the partial plans are sorted according to the metric so that the incomplete plan with minimum development overhead (with respect to the metric) is returned.

PlanSP shows the solution choices for the text mining example in Figure 20. Only using the components in the library, choice S_1 is not feasible because it violates the performance requirement of 5 time units for S_{new}. The other choice is to go for new development and S_2 describes this further. It shows that that the two sub-components of *TextMP-1* (T_1^1 and T_2^1) could be used with a new software piece developed to fill in the missing visualization need (create a hierarchical view from a partitioned cluster view). Moreover, it should not take more than 2 time units (seconds) and be integratable in applications within 7 hours or less.

that fall within the range can be returned.

```
FindBestNewDevlopmentSpec(I, G, A, CostMax, EffortMax)
    1. CurrentCost = 0; CurrentEffort = 0;
       BestMeasure = CalculateMetric(CurrentCost, CurrentEffort);
       BestPlan = {}; MinSpec = I, G
    2. If G ⊂ I
    3.   Return BestPlan // No new development is needed
    4. End-if
    5. Ninit.sequence = {}; Ninit.state = I;
       Ninit.cost = 0; Ninit.effort = 0;
    6. Q = Ninit
    7. While Q is not empty
    8.      N = Remove an element from Q
    9.      Let S = N.sequence; T = N.state
    10.     For each component $A_i$ in $A$
    11.          If precondition of $A_i$ is satisfied in state S and
                 Ninit.cost ≺ CostMax and Ninit.effort ≺ EffortMax
    12.            Create new node $N'$ with:
                   $N'$.state = Update S with result of effect of $A_i$
                   and $N'$.sequence = Append(N.sequence, $A_i$)
                   $N'$.cost = update(N.cost, $A_i$.cost)
                   $N'$.effort = update(N.effort, $A_i$.effort)
                   Measure = CalculateMetric($N'$.cost, $N'$.effort);
    13.          End-if
    14.          If G ⊂ $N'$.state and Ninit.cost ≺ CostMax
                 and Ninit.effort ≺ EffortMax
    15.            Return $N'$ // Return a plan
    16.          End-if
    17.          Q = Q ∪ $N'$ in sorted order according to Measure
    18.     End-for
    19. End-while
    20. Return FAIL // No plan was found


CalculateMetric(cost, effort)
    1. Return evaluation on the desired function combining
       cost and effort measures
```

Figure 19: *An algorithm to find the partial plan with the minimum metric measuring performance/cost and effort of the new development to reach G.*

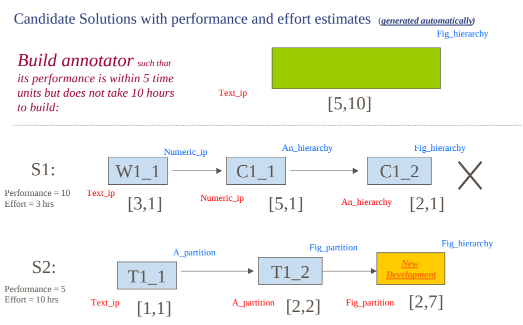


Figure 20: *Choices generated automatically with performance v/s integration effort tradeoff.*

6 Discussion and Related Work

Software reuse has been prominently investigated in software engineering. The key issue here is how to model the software components so that they could be reused with differing requirements. Our decision of representing components with predicates is essentially attribute-value based. A comprehensive study of software representation techniques in software reuse libraries on the basis of precision and recall found no statistical difference between attribute-value and other alternative representations studied but noted that the result sets could be different based on the reasoning methods used[Frakes & Pole1994].

Separation of concerns techniques like Mixin layers[Smaragdakis & Batory1998], Aspect-oriented Programming[Kiczales et al1997] and Hyperspaces[Tarr et al1999] build artifacts (aspect, layer, etc.) around the core software components so that the components could be selectively reused. Researchers have also looked at dynamic composition of object behavior based on context[Seiter et al1998] or composition policies[Truyen et al2001]. However, our main focus is a decision-support role where the aim is to automatically identify **good** candidates/ compositions (in some objective sense) for reuse rather than customize existing components during invocation. Component customization is an optional[4] and complementary feature of the reuse library that affects how the capability of the software component is published, and is useful while invoking the components. PlanSP is transparent to it because it accepts the published capabilities and reasons about their appropriateness over all possible compositions with respect to the user's goal.

To clarify the point further, it is insightful to illustrate the decision support using web services. Web services are platform and language independent software components that can be invoked over the web to fulfill some needs (or goals). An application will use a composition of web services to find the relevant set of given services and how they could be invoked (the "flow") to achieve the goals. PlanSP helps in *discovering* the right set/sequence of services while reuse work in software engineering lead to adaptation during runtime *invocation* of the services. The work closest to PlanSP is [Mudiam et al2002] where Java's JINI technology is used to discover set of services based on their architectural description. However, it does not return information about how the identified services have to be used (composition sequence) or cover maintenance scenario. Early efforts towards automatic web services composition [Ponnekanti S. & Fox2002, Srivastava2002] are also related to PlanSP.

Though the project management scenarios identified in the paper form the crux of any successful software life cycle, there is no available method that can address both of them. Moreover, although there are approaches for handling the individual scenarios, they are not automatic - the user essentially has to herself figure out the tradeoffs as shown below.

Existing methods for Scenario 1:

[4]If customization is not supported, all possible customizations can be listed in the library as separate components for the same effect. But this is clearly not a compact and desirable representation.

- Reusable software libraries exist to organize software components and facilitate their later reuse. However, the user has to manually explore them for possible candidates that can be used in their project. There is no support to the user to reason with the tradeoff between the integration time of the software component into her project and its potential performance.
- Tools like *Microsoft Project*[Microsoft1998] show timelines/ deadlines for tasks and their dependency as entered by the user. Later, the user herself has to figure out the choices among software components so that the project can be brought to timely completion.

Existing methods for Scenario 2:

- *Make* files provide a simple way to detect changes in dependent components and rebuild a software selectively based on need. But since it considers timestamp of the component rather than any change in its functionality for rebuilding the software, *Make* forces unnecessary builds that the software may not affect the software.

PlanSP is an attempt to address both the project management scenarios together. We have implemented the algorithms presented in the paper in two AI planners, ParamC for pure STRIPS domain and ParamM for metric domains (ongoing). However, the work is still early and no large scale real-world evaluation of PlanSP has yet been performed. However, even in the small examples of Section 2 presented, it is evident that the alternatives returned after automatic reasoning are non-obvious. Moreover, performance is in the order of milliseconds.

7 Conclusion

In this paper, we identified 4 common problems for which decision-support is sought during a software's life cycle and introduced a decision-support framework for software development and maintenance called PlanSP. PlanSP uses AI planning/ reasoning techniques to automatically analyze the different alternatives and assists the user in taking decisions with are **good** with respect to their metric. The novelty of the paper is in how the problems are created for tackling the specific decision-support issues and their solutions. The goal is not only to support the functionality of the software but also to respect the user's effort and performance objectives. Using examples, we demonstrated that the PlanSP framework is both useful and practical for software project management. In future, we intend to perform real-world evaluation of PlanSP on large projects and significant software libraries .

References

[Blum & Furst1995] Blum, A., and Furst, M. 1995. Fast planning through planning graph analysis. *Proc IJCAI-95* 1636–1642.

[Fox & Long2002] 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*.

[Fikes & Nilsson1990] Fikes, R., and Nilsson, N. 1990. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Readings in Planning.* Morgan Kaufmann Publ., San Mateo, CA.

[Frakes & Pole1994] Frakes, W. B., and Pole, T. P. 1994. An empirical study of representation methods for reusable software components. *IEEE Trans. on Software Engineering*, 20(8):617–630, August.

[Hall1993] Hall, R. J. 1993. Generalized Behavior-based Retrieval. *Proc. 15th Intl Conf. on Software Engineering (ICSE)*, IEEE Computer Society.

[Huff & Lesser1988] Huff, K. E. and Lesser, V. R. 1988. A Plan-Based Intelligent Assistant That Supports the Process of Programming. *ACM SIGSOFT Software Engineering Notes*, 13:97–106, November.

[Kiczales et al1997] Kiczales, G., Lamping, J., Mendhekar, C., Lopes, C., Loingtier, J., and Irwan, J. 1997. Aspect-Oriented Programming. Applications. *Proc. ECOOP'97*, June.

[Microsoft1998] Microsoft. 1998. *Microsoft Project Version 4.0 User Guide.* Microsoft Press.

[Moder & Phillips1964] Moder, J. J., and Phillips, C. R. 1964. *Project Management with CPM and PERT.* Reinhold Publ., Chapman & Hall Ltd., London.

[Mudiam et al2002] S. Mudiam, G. Gannod, and T. Lindquist. 2002. A Novel ServiceBased Paradigm for Dynamic Component Integration. *AAAI 02 Workshop on Intelligent Service Integration.*

[Ponnekanti S. & Fox2002] Ponnekanti S. and Fox, A. 2002. SWORD: A Developer Toolkit for Web Services Composition *WWW Conference*, Hawaii.

[Pressman1996] Pressman, R. 1996. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, 4th Edition, ISBN 0070521824.

[Seiter et al1998] Seiter L., Palsberg, J., and Lieberherr, K. 1998. Evolution of Object Behavior using Context Relations. *IEEE Trans. on Software Engineering*, 24(1).

[Smaragdakis & Batory1998] Smaragdakis Y., and Batory, D. 1998. Implementing Layered Designs with Mixin Layers. *Proc. ECOOP'98.*

[Srivastava2002] Srivastava, B. 2002. Automatic Web Services Composition Using Planning. *Proc. Knowldge-Based Computer Systems (KBCS)*, Mumbai, pg 467–477.

[Tarr et al1999] Tarr, P., Ossher, H., Harrison, W., and Sutton, S. 1999. N Degrees of Separation: Multi-Dimensional Separation of Concerns. *Proc. ICSE'99.*

[Truyen et al2001] Truyen, E., Vanhaute, B., Joosen, W., Verbaeten, P., and Jorgensen, B.N. 2001. Dynamic and Selective Combination of Extensions in Component-Based Applications. *Proc. 23rd ICSE.*

[Weld1999] Weld, D. 1999. Recent Advances in AI Planning. *AI Magazine*, Volume 20, No.2, pp 93-123.