A Planning-based Decision-support Tool for Software Project Management

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

EXTENDED ABSTRACT

Problem: 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. Though tools exist to track component dependencies and historical changes, the key software management hurdle is the manual evaluation of the trade-offs in leveraging current components v/s investing in new software development.

Consider a typical software project management scenario. The requirement specification is acquired from the customer and then, the solution architects devise the Work Breakdown Structure (WBS)(Moder & Phillips 1964) of the problem to identify the different tasks at some granularity. This information is input to a project management tool like Microsoft Project along with estimates on time and resources for each task. The tool may have 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.

Solution Approach: We propose a framework called PlanSP to analyze different project management choices *automatically* as following reasoning problems and thereby assist the user make informed decisions.

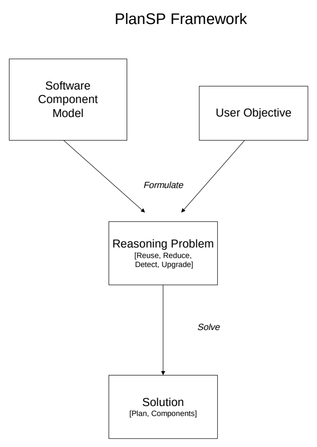

Figure 1: *PlanSP approach of decision support in software project management.*

- 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}.

  [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 reasoning problems related to planning (see Figure 1). We cast `Reuse` as the problem of finding components (actions) relevant to the goals of the software to be built while `Reduce` is formulated as a (cost-based) plan generation problem. `Advise` is seen as the enumeration of components (actions) in a project implementation (i.e., plan) while `Upgrade` is solved by a combination of solutions for the `Reuse` and `Advise` problems. Due to space limitations, the details are omitted(Srivastava 2003).

The insight in posing a software project as a planning problem is that the initial state I can be seen as the input data specification while the 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. The plan for realizing the goal(s) is the Work Breakdown Structure (WBS) of the application. Therefore, during software development, we reason about how best to arrive at a cost-effective WBS while during software maintenance, a WBS is given and we reason about any impact on it due to internal events (upgrade of existing components) or external opportunities (new components being released).

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.

The software model is based both on the structured information that is available about a software at its release time (like dependency information used by $Make$), represented as predicates, 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. Predicates are essentially attribute-value representation which a comprehensive study of software representation techniques in software reuse libraries found not statistically different from other alternative representations studied(Frakes & Pole 1994). The formal model for software components is stored and appropriately referenced by PlanSP.

The novelty of our work is in how the reasoning 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.

PlanSP is ideally suited for software applications complying with Service-Oriented Architecture (SOA) principles, e.g. web services. In SOA, the description of a software component's inputs and outputs (functionality, dependencies, types and metrics) is published to a service directory (e.g., UDDI) in a description language like WSDL while services are themselves deployed on a server. An application finds about the published services from the directory, chooses a set of interest, and invokes them using the service specification. The tool is useful for general software projects as well, as long as the description of the software components can be explicated.

Related work: There is no available method that can address all the identified project management scenarios. During software development, tools like *Microsoft Project* 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. Build tools like *Make* provide a simple way to detect changes in dependent components based on timestamps. However, *Make* can force unnecessary builds of components that may not affect the software functionality.

There has not been much use of automated reasoning techniques in software reuse and maintenance. Previously, (Huff & Lesser 1988) 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. Existing separation of concerns techniques like Mixin layers, Aspect-oriented Programming and Hyperspaces build artifacts (aspect, layer, etc.) around the core software components so that the components could be selectively reused but the selection process is user-driven.

Current status: We have implemented PlanSP using two AI planners, ParamC for pure STRIPS domain and ParamM for metric domains and it has been tested for software components in the Natural Language Understanding domain containing software components in multiple human languages (a total of 20-odd components) with various metrics(Srivastava 2003). As noted earlier, PlanSP is ideally suited for software applications in the Service-Oriented Architecture where component specification are readily available but it is also useful for general software projects as long as the description of the software components can be explicated. In future, we plan to conduct large scale real-world evaluation of PlanSP in the context of web services and/or java beans (IBM Websphere). However, even in the small examples presented, it is evident that the alternatives returned after automatic reasoning are non-obvious.

References

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.

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.

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

Srivastava, B. 2003. PlanSP: A Framework to Automatically Analyze Software Development and Maintenance Choices. *IBM Research Report RI02025. Available at http://domino.watson.ibm.com/library/CyberDig.nsf/Home*