Component-oriented Programming as an AI-planning Problem

Debasis Mitra and Walter P. Bond
Department of Computer Sciences
Florida Institute of Technology
150 West University Blvd.
Melbourne, FL 32901, USA

Abstract

A dream of the software-engineering discipline is to develop reusable program-components and to build programs out of them. Formalization a type of component-oriented programming (COP) problem (that does not need any non-trivial effort for gluing components together) shows a surprising similarity to the problem of Planning within the Artificial Intelligence (AI). This short paper explores the possibility of solving COP by using AI-planning techniques. We have looked into some closely related AI-planning algorithms and suggested directions on how to adopt them for the purpose. Other important related issues like the target specification languages and other relevant research disciplines are also being touched upon here.

Key Words: Planning; Component-oriented programming; Intelligent software-engineering

1. Introduction

The success story of modern engineering lies with the capability of designing any target object using well-understood components. Unfortunately this is still mostly a dream within Software Engineering. As a result, programming is still considered as primarily an art. A successful reuse of the software components - as a standard practice - would transform this area from an art to an engineering discipline. A preliminary approach in this direction was to develop the standard library of routines and add it to the environment of a programming language. Common Lisp and C languages have used this technique quite extensively. In the recent years C++ has extended the technique by standardizing its template library (STL) that contains a repertoire of higher-level objects. JAVA language from its inception has incorporated a huge set of API's in a very organized way. However, the concept of reusable components demands an even higher level of architecture. A recent movement toward that direction comes from the introduction of the JAVA Beans. It provides the programmer to store and reuse components in a nice environment. However, the maintenance of the library of components is almost left to the user programmer. Such environments (like BDK or Bean Development Kit) are provides very little help for writing a program by utilizing such components. This short paper proposes a framework for intelligently usage of components for the purpose of developing a program.

There are some domains that expect the programs to be developed out of the existing components only. Numerical computation, particularly in the scientific and engineering area, is a domain of this nature. Often the user (who is typically a scientist or an engineer) writes a script code developing a model for computation that he or she wants to perform. The script code is nothing but an ordering of some of the library routines, with some parameters instantiated within the latter ones. An example of such numerical computation is done in the area of seismic data processing, typically within the petroleum exploration-industry. In that domain a script code is being written for ordering some signal processing and data handling routines from a software library. The script code is subsequently pre-processed for developing a program (which primarily calls the library routines) for processing data.

The types of domains described above deploy an extreme situation of component-oriented programming, where the "glue" codes between components in a program are mostly trivial and the pre-processor automatically generates them. However, even in this situation the user is expected to know a great level of details about the individual routines as well as their relationships with each other. The interface modules (graphical or otherwise) or the pre-processors do not provide much help to the user in writing the script code or in checking its consistency. In a nutshell, these domains deploy "component-oriented programming" but do so quite manually. Our proposal is to apply the AI-planning techniques for the purpose of automatically developing programs using reusable components from a library.

Planning is a core area within the Artificial Intelligence (Russell and Norvig, 1995). The central problem there is to order (partial or linear) some operators from a given set, in order to achieve some goal. World states are represented in some language. A finite number of primitives are provided as a starting state of the world and the goal state of the world. Both are described in the same language. Each operator has a set of preconditions and a set of effects or post-conditions, again both are described using the same Planning-language. Planning algorithms search through the operator set to develop a sequence (or a partial order) of some of the operators in order to change the world from the input start-state to the input goal-state. In the AI-planning problem, if we replace the operators with some program-components, then the same problem could be viewed as a program development-problem by strictly using components. Planning is a heavily researched area and some good progress has been made in the last few decades that could be taken advantage of towards the component-oriented programming. This concept paper explores that possibility.

Section 2 defines the problem formally. In a following section we describe the Planning problem and mention a few related AI-planning techniques. In section 4 we discuss a feasible software specification language that may be modified toward an AI-planning language for our purpose. Section 5 puts forward a proposal for modifying some existing techniques for solving the component-oriented programming problem. A short section 6 mentions some other research works where the developed methodologies are strongly related to the proposed framework here and so, has a strong relevance in the research. The paper is concluded in the following section.

2. The problem definition

A component-library constitutes a set of *components* $C = \{c_1, c_2, \ldots\}$. Each component c_i is recursively defined as either a simple element from a set S of *software pieces*, or an n-ary directed graph {V, E}, where the nodes in V are components from C, and E is a set of n-ary edges between some nodes. The semantics of an edge could be a simple component hierarchy, or could be a temporal linkage (before/after chain) between the sub-components within the component. Each component also has a specified *required-environment* r_{ci} needed for its existence in a solution (program), and creates a *target-environment* t_{ci} after it is executed. Each of these environments could be as simple as the input/output parameters list for a component/subroutine. Each c_i is also associated with a set of properties $p_i = \{p_{i1}, p_{i2}, \ldots\}$.

The component-oriented programming problem, in our restricted sense, is COP=(A, R, T). The problem is to elaborate (or follow) a given software architecture A by creating a graph with nodes c_i's from C, when some input-environment R and target-environment T are being provided apriori, such that the whole target-environment T is satisfied. The architecture A itself may be a graph with its nodes being a higher-level nodes, or may be a broad-level description, using the language that is used for specifying the sets of properties p_i's of the components. Architecture A is a list of predicates that needs to be satisfied in a solution by the p_i's of the components. Nodes in the architecture A may be instantiated from C in such a way that the target environment T is satisfied. Initial environment R can be used to satisfy the components' required environments r_{ci}. Also, a component c_1's output t_{c1i} can be used to satisfy another component c_2's required-environment r_{c2i}, when c_2 is located *downstream* in the solution compared to c_1. A solved COP may be subsequently added to the library C as a newly added component.

Example: A data processing situation using filters. S is a simple suit of filter routines, C is a library of elements of S and some other composite filters developed by solving problems. Each c_i can be applied under some assumption (r_{ci}) about the data that needs to be processed, and will produce some quality (t_{ci}) in the filtered data. Each filter c_i also has a description $p_i = \{p_{i1}, p_{i2}, \ldots\}$. A problem instance COP is a broad architecture (A) about the requirement of the type of filters needed in a specific order, some information on the quality of the input data (R) and the required quality of the output data (T). A solution will be appropriately ordered (linear or partial) chain of components from the library C for achieving the required data-quality T such that "chain" follows the input broad-level architecture A for the solution.

3. Relation to the AI-planning

A Planning problem (Russell and Norvig, 1995) involves a library of operators $C=\{c_1, c_2, \ldots\}$, each c_i has a set of preconditions r_{ci} and a set of post-conditions t_{ci}. A planning problem P=(R, T) with a set of start-state predicates R and goal-state predicates T, is to create a directed graph A out of the operators from C such that all elements of T are achieved. All of R, T, r_{ci}, t_{ci} come from a set of world state predicates specified in a Planning-language. As posed here, P is a sub-problem of COP described in the previous section where the architecture/descriptions A is a *null* graph. Also, descriptions p_i for each component c_i is a *null* set in the planning problem.

Actually COP problem is an easier version of the planning problem because of the extra constraints in A. Existence of a non-null initial A can be considered as a plan shell to start with. Additional descriptions p_i's in c_i's help in the problem-solving process. In both the problems the objective is to develop a final graph A. In this sense COP problem is a sub-problem of P, where P is enhanced with additional input A and p_i 's. Hence, both the problems COP and P are equivalent to each other.

Some of the planning strategies that we can preclude for COP are dynamic planners, conditional planners or reactive planners (Russell and Norvig, 1995). These types of planners involve a capability to change the plan at run time which is not warranted in the COP problem. The later problem is very much of a static nature. A first pass observation suggests that we need a partial-order planning (Weld, 1994) because the resulting plan need not be a linear chain of components in COP. Secondly, we need to deploy some type of hierarchical planning for utilizing some component-hierarchy that may be available in the component library C.   Below, we will discuss some of the existing planning frameworks that suit the COP problem.

GraphPlan (Blum and Furst, 1997) generates a partial-order plan where one or more operators are allocated in each time-step, total plan being a sequence of a finite number of time-steps. The algorithm works by propagating mutual-exclusion constraints between instantiated operators (and state-predicates) from one stage to the next, that prevents operator being "chained" next to each other. The graph plan could be adopted toward hierarchical plan generation for solving the COP problem. Extension of the GraphPlan has been made by Do and Kambhampati (2001) by using constraint satisfaction (CSP) approach. Thus, the propagation of mutual-exclusion constraints could be made more efficient by using different heuristics like dependency-directed back-jumping or forward-checking etc.

Consider a somewhat more specific scenario of the example in the previous section. There are two sets of filter routines existing within the component library, one for onboard a satellite and another for using on ground (in an online satellite-generated data processing scenario). A simple architecture (A) is given in a problem instance that states that the ground filters should follow the satellite filters. This constraint could be translated as mutex constraints between the operators while solving the problem. These mutex constraints between components are not static as considered in the “GrpahPlan,” but are created dynamically for the problem instance (in A/p_i's). In case in the initial architecture A is reversed, i.e., the input description requires that the satellite filters should follow the ground filters, then the mutex constraints would be reversed and a different solution will be achieved accordingly.

O-Plan (Tate, 96) is the Open Plan Architecture in which hierarchical planning could be done with a mixed-initiative between the system and the human. This feature is useful for component-oriented programming with programmer interaction. Also, the framework provides opportunity for incorporating “descriptions” as in ‘A’ and p_i’s in the COP. For these reasons O-Plan is also a closely related framework to our problem, even though it is developed primarily for a dynamic environment, e.g., emergency management. A particularly interesting model within the O-Plan is the <I-N-OVA> model, where the planning is done by using constraint manipulation and where the design rationale is being captured systematically. Formalizing such “design rationale” could constitute a step forward toward handling the “descriptions” as in ‘A’ and p_i’s in the COP.

4. Requirement/Component specification language

One of the major problems in this work is to develop a software architecture-specification language. The language should be able to describe the components (p_i, r_{ci}, t_{ci}) and the COP problem (A, R, T). It also needs to be a Planning language at the same time. Hence, it must be a hybrid between two such languages.

Currently the most popular object-oriented software architecture-specification language is the universal modeling language or UML. It is a standardized visual language with semantics attached to the icons. While the visual interface could be very useful for specifying component-based architecture of a target program, the propositional nature of UML makes it somewhat inflexible (cannot handle first-order formulas) and thus, not so suitable as a target for adopting AI-planning languages. Most of the current-generation Planning languages are based on the first-order predicate logic that allows a better expressiveness.

A good candidate for the functional specification of component-oriented architecture of any software is the Z-notation (mainly from the Programming Research Group at the University of Oxford, see in Shaw and Garlan, 1996). Z-notation has a first-order type syntax and semantics. All the three aspects of the COP problem, A/p_i, R/r_{ci}, and T/t_{ci}, could be described in such a modified Z-language.

5. A scheme for doing component-oriented programming

In Mitra (2001) we have proposed a relational data model-based algorithm for helping the programmer to choose appropriate components (stored in a relational database) for an input architecture of the target software. The algorithm (CPRAO) does constraint propagation (very similar to the one required in the map-coloring problem) utilizing relational algebraic operators like project and join, relevant to the relational data model. At each stage of the iteration it lets user instantiate an element in the architecture (or a node in the graph) out of a set of valid components suggested by the algorithm, and then propagates the constraint (by utilizing relational operators) in order to filter the set of valid components (domains for the nodes) on the adjacent nodes. For a chosen target problem domain of component-oriented programming, we have used matching of the set of input/output parameter lists of adjacent components as the basis of constraint propagation. In the COP problem as stated in this paper, these i/o parameters could be considered as the set of required-environments r_{ci} and the set of target-environments t_{ci} for each component. The CPRAO algorithm of Mitra (2001) could be adopted for constraint propagation within a planning algorithm. A practical advantage of the CPRAO algorithm is that it allows us to use a back-end relational database for storing the component library.

The satisfaction of the initial architecture/descriptions A in the COP problem has to be achieved by the set of descriptions p_i's of the instantiated components. This can be done by deploying a technique like the unification/resolution as in the Logic Programming. Hence our scheme is to combine two different schemes of search: (1) Planning, for the purpose of chaining the required and target environments of the components by choosing appropriate components, and (2) Logic Programming, for the purpose of satisfying the original description in a first-order like requirement specification language (used for specifying A and p_i's). The two search techniques will complement each other and thus, enhance the aggregate efficiency of the algorithm.

Example of such combination already exists in the TAL or the Temporal-action logic Planner from the Linkopings University-group in Sweden (Kvarnstrom and Doherty, 2001), which apparently is one of the fastest running Planner at this moment. In this planning language each of the planning operators is optionally extended with two parameters t1 and t2, indicating respectively the start and the end times of the intervals for applications of the operators. The planning algorithm uses these two special variables for unification/resolution directing the search for satisfying the temporal constraints. Our scheme is to use the same technique as that used in the TAL for the purpose of satisfying the architectural description A with the properties p_i's of the components/operators, instead of satisfying only the temporal constraints as in TAL.

The scheme is to run the GraphPlan algorithm (Blum and Frust, 1997) for the COP-problem, or the GraphPlan's extension GP-CSP algorithm (Do and Kambhampati, 2001). At each stage of elaborating the Plan Graph in the algorithm, the scheme will use the component-descriptions p_i in order to satisfy the input descriptions A, which either leads to pruning of some branches (components) of the search tree or to guide the search toward the maximal satisfaction (by choosing one or more components). This step will deploy some type of resolution algorithm as in the TAL planner. The mutual-exclusion-relationships needed by the GraphPlan algorithm will be created by running the CPRAO algorithm over the environment parameters.

6. Related works in other areas

(1) There are attempts to develop databases for programs (Paul and Prakash, 1996). This type of work typically addresses whole programs and not components. However, the ontology developed for describing a program for the purpose of designing a database is relevant to our work. (2) In a workshop on the area of Intelligent Software Engineering at AAAI-1999 conference Fischer and others from the NASA Ames Research Center have proposed a framework for retrieval and adaptation of components. Their work in this area is a close parallel to our proposed scheme. (3) The CAD (computer-aided design) area is also involved with the type of domains we are targeting. Components in CAD software simulate geometrical objects. Such a software is supposed to allow a user to assemble components in a drag-and-drop oriented visual environment in order to develop a model for subsequent construction (and also possibly for simulation of its behavior). Constraint processing is a strongly relevant issue there and schemes have been proposed (Bhansali and Hoar, 1998) for the purpose. The proximity of the CAD and the COP areas is self-evident. (4) Memon et al (2000) have tried to develop a mechanism for generating test-cases for testing functionalities of graphical user interfaces. They utilized an AI-planning approach that is somewhat similar to that proposed here for solving the COP-problem.

7. Conclusion

In this short paper we have proposed a framework for doing component-oriented programming automatically (or semi-automatically) using the Planning techniques from the artificial intelligence area. The work is primarily based on the observation that the two problems, COP and AI-Planning, are very similar to each other. We have formulated the two problems in a similar framework here to show this symmetry. We have identified some planning techniques and proposed how to modify them to solve the COP problem. We expect that such modifications to the AI-planning algorithms will actually enhance the efficiency of those algorithms, because the additional information (description or architecture of the required solution) will provide more guidance to the search procedure. Ostensibly, this also points to a new direction in the AI-planning research. We have also touched upon some related works in other areas.

References

Blum, A. V., and Furst, M. L., (1997). "Fast planning through planning graph analysis," Artificial Intelligence journal, Vol. 90, pp. 281-300.

Bhansali, S. and Hoar, T.H., (1998). Automated software synthesis: An application in mechanical CAD. IEEE Transactions on Software Engineering, 24(10).

Do, M. B., and Kambhampati, S., (2001). "Planning as constraint satisfaction: Solving the planning by compiling it into CSP," Artificial Intelligence journal, Vol. 132, No. 2, pp. 151-182.

Kvarnstrom, J., and Doherty, P., (2001). "TAL planner: A temporal action logic based forward chaining planner," Annals of Mathematics and Artificial Intelligence journal.

Mitra, D., (2001). "Interactive modeling for batch simulation of engineering systems: A constraint satisfaction problem," Lecture Notes in Artificial Intelligence (Proceedings of the IEA/AIE-2001), Vol. 2070, pp. 602-611, Springer.

Momon, A. M., Martha, E. P., and Soffa, M. L., (2000). "Plan generation for GUI testing," Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2000), pp. 226-235, Breckenridge, Colorado, USA.

Paul, S. and Prakash, A., (1996). A query algebra for program databases. IEEE Transactions on Software Engineering, 22(3).

Russell, S. and Norvig, P., (1995). Artificial Intelligence: A Modern Approach. Prentice Hall, Englewood Cliffs, New Jersey.

Shaw, M., and Garlan, D., (1996). "Software Architecture: Perspectives on an Emerging Discipline," Printice-Hall, Inc., New Jersey, USA.

Tate, A., (1996). "Representing plans as a set of constraints – the <I-N-OVA> model," Proceedings of the AIPS-96 conference, Edinburgh, UK.

Weld, D., (1994). "An introduction to partial-order planning," AI Magazine, AAAI Press.