A Plan-based Intelligent Assistant That Supports the Software Development Process

Karen E. Huff and Victor R. Lesser

Computer and Information Science Department
University of Massachusetts at Amherst

Abstract: We describe how an environment can be extended to support the *process* of software development. Our approach is based on the AI *planning* paradigm. Processes are formally defined hierarchically via plan operators, using multiple levels of abstraction. *Plans* are constructed dynamically from the operators; the sequences of actions in plans are tailored to the context of their use, and conflicts among actions are prevented. Monitoring of the development process, to detect and avert process errors, is accomplished by *plan recognition*; this establishes a context in which programmer-selected goals can be automated via *plan generation*. We also show how *non-monotonic reasoning* can be used to make an independent assessment of the *credibility* of complex process alternatives, and yet accede to the programmer's superior judgment. This extension to intelligent assistance provides deeper understanding of software processes.

1. Introduction

Environments have traditionally provided minimal support for the *process* of software development. Separate parts of the process are typically supported by separate tools, while global patterns of tool usage are not made explicit, and are not exploited. Thus, the environment cannot prevent a programmer from starting compilations before an appropriate context is set up, enforce a policy of regression and performance testing before a customer release, insure that new source versions are checked back into the source code control system, or guarantee that source files are deleted only after their contents have been archived or superseded. Such support would be valuable to both programmers and their managers.

Extending environments to incorporate process support requires explicit representation of the software process, showing how software development *goals* are mapped into sequences of environment *actions* (Figure 1). Typical goals (during implementation) are concerned with adding functionality to a system version, fixing bugs, adhering to various project-specific policies, and maintaining an organized on-line workspace. The actions available to achieve these goals consist of invocations of tools provided within the environment. Knowledge of a specific software process governs the mapping from goals to actions, distinguishing between appropriate sequences of actions and random ones.

This view of software processes fits the *planning* paradigm, an AI approach to a theory of actions. In planning [9], knowledge of a domain is expressed in *operators* (parameterized templates defining the possible actions of the domain) together with a *state schema* (a set of predicates that describe the state of the world for that domain). *Goals* are logical expressions composed of the state predicates. A *plan* is a hierarchical, partial order of operators (with bound

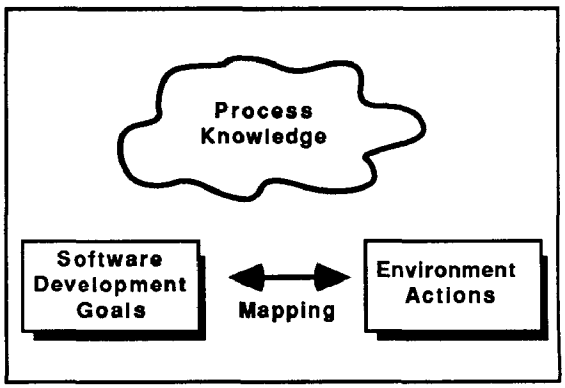

Figure 1: A View of Software Processes

This work was supported by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, and the Air Force Office of Scientific Research, Bolling AFB, DC 20332 under contract No. F30602-C-0008, supporting the Northeast Artificial Intelligence Consortium (NAIC).

Recommended by: *Robert M. Balzer*

parameters) that achieves a goal given an initial state of the world. There are two mapping algorithms: *planning*, where a plan is constructed given a goal and an initial state, and *plan recognition*, where a plan and its goal are inferred given a sequence of actions and an initial state.

When the planning paradigm is applied to software processes, operators are the vehicle for formally defining processes; plans are the data structures that represent instantiations of processes; and, assistance can be active (via planning) or passive (via plan recognition). If the programmer retains the initiative for performing the process, issuing commands exactly as at present, plan recognition can detect and avert process errors. This "kibitzing" is an automated version of a colleague watching over the shoulder of a programmer at work. Alternatively, the programmer can state a goal to be satisfied, invoking plan generation to automate achievement of the goal. Depending on the scope of the goal, plan generation may be completely automatic, or cooperative [6] (relying on interactive input from the programmer).

We call this combination of volunteered advice and cooperative automation *intelligent process assistance*—an approach to *machine mediation* of software development [2, 18] as it applies specifically to the development process. Another type of plan-based intelligent assistance, directed at understanding the deep structure of code, is described in [14, 25].

Comprehensive support of the software process requires involvement in both the complex decisions as well as the mundane details. Formally representing some of these decisions is a challenge (independent of the choice of process specification formalism). What are the criteria for choosing the baseline from which to develop a new system version, selecting tests to run, or deciding which system version is releasable? If a process assistant lacks knowledge to address these decisions, it cannot independently critique a choice made by the programmer, nor can it suggest a restricted set of likely candidates from which the programmer can choose; the level of assistance is seriously restricted.

While universal decision procedures embodying these complex criteria seem unattainable, rules can be given to cover typical situations and anticipated exceptions. Reasoning with these rules is inherently imprecise; the conclusions are plausible, but fallible—assumptions may have to be revised. By introducing *non-monotonic reasoning*, it is possible to formalize additional process knowledge to evaluate the *credibility* of alternatives for decisions that could not otherwise be addressed. The additional discrimination power is flexible, not absolute; it is possible to defer gracefully to the programmer's judgment, integrating the implications of that judgment into a revised set of assumptions.

In Section 2, we show how the planning paradigm applies to software development, giving examples of software process operators and plans. Planning is compared with other approaches to specifying and supporting software processes. We also show how a plan-based intelligent assistant is integrated into an environment architecture, and give examples of volunteered advice and automation. In Section 3, we present an approach to deeper process

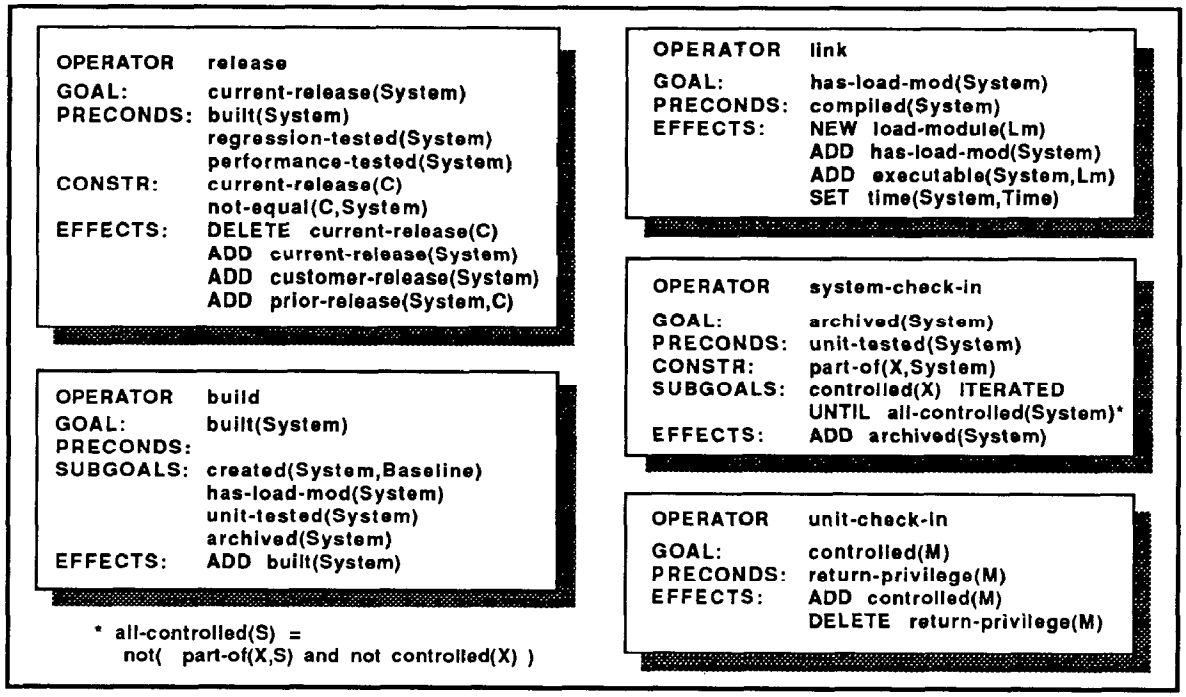

Figure 2: Software Process Operators

understanding. Additional process knowledge is expressed in monotonic and non-monotonic rules, and a *truth maintenance system* is used to reason with this knowledge. In the final section we summarize the GRAPPLE project status and describe future extensions.

2. Plan-based Process Support

2.1 Software Process Operators

As an example of a software process, consider how implementation might be carried out for a traditional programming language such as C, assuming currently accepted engineering practice such as incremental development, source code control, bug report database, and specialized test suites. A partial library of operators for this domain appears in Figure 2. A state schema supporting these operators is sketched in Figure 3, using the ER model of data [5] as the graphical presentation; relationships and attributes correspond to the logical predicates used in the operator definitions.

Operator definitions follow the state-based, hierarchical planning approach [22, 23, 28]. Each operator has a *precondition* defining the state that must hold in order for the action to be legal, and a set of *effects* that defines the state changes that result from performing the action. These core clauses are augmented by a *goal* clause that defines the principal effects of an action (thus distinguishing them from "side-effects" of the action), and a *constraints* clause that defines restrictions on parameter values. The *unit-check-in* operator in Figure 2 describes the action of checking a new version of a source module into a source code control system (such as RCS [24]). The precondition on the action requires that the module was previously checked-out with return privileges. The goal of the action is that the new version is now "under" source code control; this is also one of the effects of the action. There is one side-effect, namely that the return privilege is lost (another operator is used to describe the type of check-in that retains the return privilege).

An abstraction hierarchy is created through *complex* operators having *subgoals*. (This is essential for describing complicated processes.) The subgoals of the build operator decompose building into four parts: creation of new source versions, creation of a load module, running unit tests, and checking-in. *Primitive* operators, which do not have subgoals, correspond to the atomic actions in the domain. Some primitive actions, like *unit-check-in*, correspond to tool invocations (perhaps with specific parameter settings). Other primitive actions correspond to command-language scripts; *release* could be implemented as a script that copies a load module to the customer's directory and issues a release notice.

All the policies and procedures that are associated with a particular software development process must be included in the operator definitions. Changing the definitions of the operators changes the process that will be followed. For example, the *release* operator in Figure 2 requires that performance tests and regression tests be run before a customer release is made. These requirements could be relaxed (by deleting one or both testing-related preconditions) or strengthened (by adding another precondition requiring execution of a particular analysis tool or updating of release documentation).

2.2 Software Process Plans

Plans are constructed dynamically by instantiating operators. Operator definitions contain sufficient

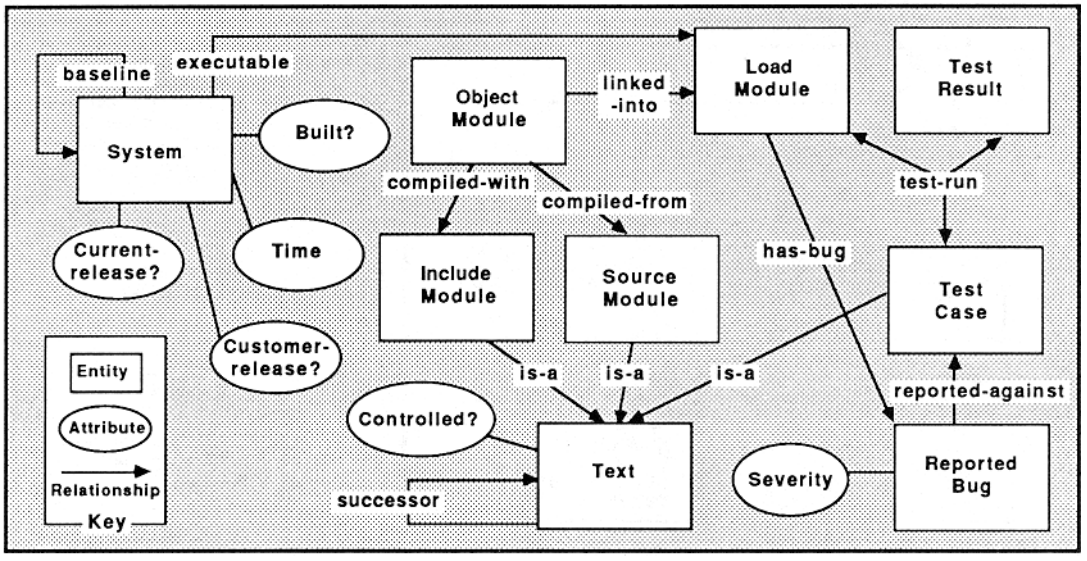

Figure 3: Software Process State

information to reason about sequences of actions without actually executing the actions. The state changes that an action causes are explicit; therefore, sequences of actions can be "simulated". The exact preconditions for an action are explicit; therefore, actions can be ordered correctly. Concurrency is implicitly allowed, subject only to the stated preconditions. Plans are automatically tailored to the exact context for which they are needed. If a subgoal has already been achieved as a side-effect of prior activity, actions to achieve that subgoal will be omitted. If part of a plan fails during execution, replanning will fill the gaps left by the failure (and only those gaps).

Planning is distinguished from other theories of actions by its emphasis on goals to achieve, not actions to perform. Determination of actions proceeds from goals, so that contingency handling (e.g., for redundancy or failures) is internal—not external—to the planning system. When process definition is procedural [19] or event-based [11], the composition and ordering of actions is predetermined; any contingency handling must be built into the definition by hand, in advance. A behavioral approach to process modeling that combines action and goal orientations is described in [29].

A partial example of a hierarchical plan is given in Figure 4. There, a vertical slice covering three hierarchical levels is shown; lower levels reveal more detail in the plan. Downward arrows between levels connect desired states with operators instantiated to achieve them; in general, there may be several alternative operators that could achieve a given state. Arrows within levels show how the achievement of certain states is partially ordered with respect to time. These temporal constraints follow from the operator definitions: precondition states must always precede subgoals, for example.

A special type of contingency arises when there are conflicts among actions. For example, consider an instantiation of the build operator that involves changing one source module. The new source module must be preserved from the time it is created until it is checked-in. An action such as editing (that would contribute to building the next system version) cannot be allowed to interfere with accomplishing the check-in. Strategies for salvaging the plan include preventing editing until check-in occurs or inserting an action to make another copy of the module. Conflicts can occur within a single plan as well as between two plans being carried out concurrently. The simple action of deleting something has the potential of interfering with almost any active plan.

Planning algorithms handle conflicts via domain-independent methods. Associated with each precondition or subgoal is a *protection interval* that begins when the precondition or subgoal is achieved and ends when the operator begins (for preconditions) or ends (for subgoals). A conflict occurs when a precondition or subgoal is

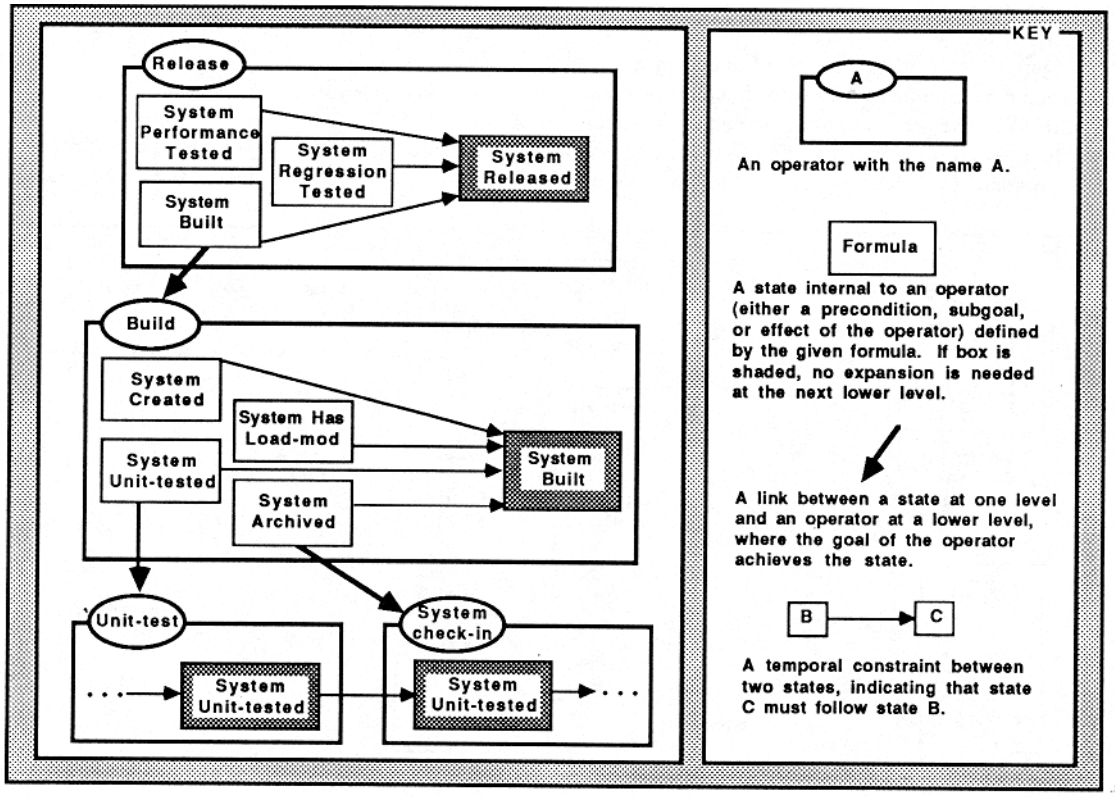

Figure 4: Vertical Slice of Hierarchical Plan

destroyed during its protection interval. Domain-independent methods for resolving conflicts, described in [12, 22, 23, 28], include imposing additional temporal constraints and selecting alternative operators.

This ability to anticipate and prevent conflicts (among operators that are fundamentally *if-then* rules) distinguishes planning from other rule-based systems. Marvel [15] uses forward and backward chaining to automate chores that are prerequisites to or consequences of programmer actions. Glitter [8] cooperatively automates goals in the transformational development process. These systems do not prevent conflicts among rules. Systems based on *<condition, action>* rules, such as CLF [1] and Genesis [21], lack descriptions of effects; they cannot reason about the consequences of actions, and therefore cannot prevent conflicts. Planning techniques are used in two existing systems. Agora [4] provides a domain-specific planner for tasks relating to heterogeneous, parallel systems. Although help systems usually provide advice that is too local to qualify as process support, UC [27] now uses planning to gain a more global perspective [17].

2.3 Integration in an Environment

The integration of an intelligent assistant into an environment is shown in Figure 5. The base environment contains a command language processor that invokes the available tools. The tools in turn reference and update the environment storage system, implemented as a database [3, 10, 20] or objectbase [26], comprising not only software products but also their attributes and the relationships among them.

The intelligent assistant has an active component based on algorithms for plan recognition and plan generation. These algorithms are process independent; they obtain process knowledge by referencing a library of operators. In order to construct plans, the algorithms must reference the current state of the world, which is essentially the database already present in the environment. The intelligent assistant also contributes additional state information to this database. For example, the effects of the *release* operator define the "meaning" of release from a higher-level perspective than the script implementing release. The script invokes the copy command to copy the contents of one file to another; copy is not aware that a release is being copied or that the release sequence is being extended.

The programmer communicates with the intelligent assistant, which moves smoothly between passive and active assistance. For commands issued in the normal way, plan recognition is invoked to find an interpretation for the action. If a valid interpretation is found, the command is passed to the command language processor for execution. An *interpretation* is a path from the action up through the plan hierarchy to a top-level operator of an existing or new plan; an interpretation is *valid* if all relevant preconditions and constraints on the path are satisfied. When the programmer issues the command *achieve <goal>*, plan generation is invoked to produce a sequence of commands for execution by the command language processor. Plan generation can also be invoked to satisfy a precondition to make an interpretation valid. Specific examples follow.

2.4 Process Support

Consider the plan recognition situation diagrammed in Figure 6. Here, the programmer has been building a new system version (S1), and work has proceeded as far as unit-testing. Thus, three subgoals of *build* (at level 1) have been satisfied (the detail of exactly how this was accomplished has been omitted). The current state of the world is also diagrammed, showing that S1 was constructed from two source modules and one include module. Assume that the programmer has just issued a command for *unit-check-in* on module C. This action is "recognized" as fitting into the plan for building S1, because *unit-check-in* (on level 3) satisfies a subgoal in *system-check-in* (on level 2) which satisfies the remaining subgoal in *build* (on level 1). This interpretation for *unit-check-in* of C is valid, because all necessary preconditions are met: return privileges exist for C (level 3) and S1 has been unit-tested (level 2). With a valid interpretation, no advice need be volunteered.

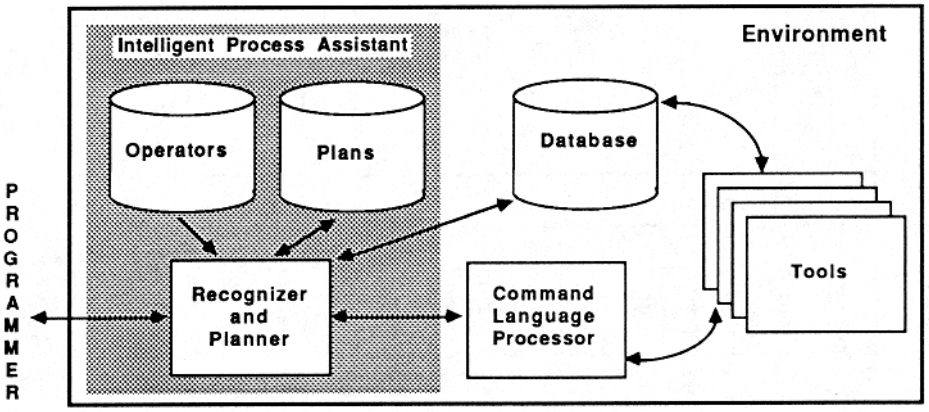

Figure 5: Environment with Intelligent Assistant

Since the roles and interrelationships of actions are explicitly represented in plans, agendas (what needs doing) and status reports (what has been done) can be generated at multiple levels of abstraction. A plan can be constructed for any agenda item, and the rationale provided for any completed activity. In contrast, the DSEE task management facilities [16] allow users to associate actions performed with a task/subtask structure, but any intelligent processing of stored task information must be provided by the programmer.

Three additional recognition examples are:

(1) If no return privileges exist for C, then the interpretation of *unit-check-in* of C is not valid due to an unsatisfied precondition at level 3. Plan generation can be invoked to satisfy the precondition, thereby making the interpretation valid.

(2) If the programmer had issued a command for *unit-check-in* on module B, this action would be recognized as superfluous—its goal is already true (presumably, B was not modified to make S1).

(3) If the programmer had attempted to relinquish the return privileges on C (perhaps meaning to do a check-in but garbling the parameters on the command), the action could be "doubly" an error. Return privileges on C are necessary to doing *unit-check-in* of C, which is necessary to completing the *build* of S1. Since the proposed action interferes with other actions, the programmer is asked to confirm it before execution (assuming the interpretation is otherwise valid). However, there may be no valid interpretation for this action (e.g., no "reason" to do a relinquish); then an error would be reported.

As an example of how planning and plan recognition are complementary, consider the request *achieve built(S1)*. The goal is that of the partially completed *build* plan, so the request equates to completing that plan. If this happens after unit-testing is complete, there will be one remaining subgoal in *build* to satisfy. The *system-check-in* operator (level 2) will be chosen to satisfy it, and *unit-check-in* (level 3) will be chosen to expand two subgoals in *system-check-in*; since B is already checked-in, the third subgoal is vacuously satisfied. Because the programmer does not have return privileges for A, a further expansion of the precondition in *unit-check-in* for A will have to be done. The final plan consists of three actions: two *unit-check-ins*, one of which is preceded by a dummy *check-out* to acquire the return privilege.

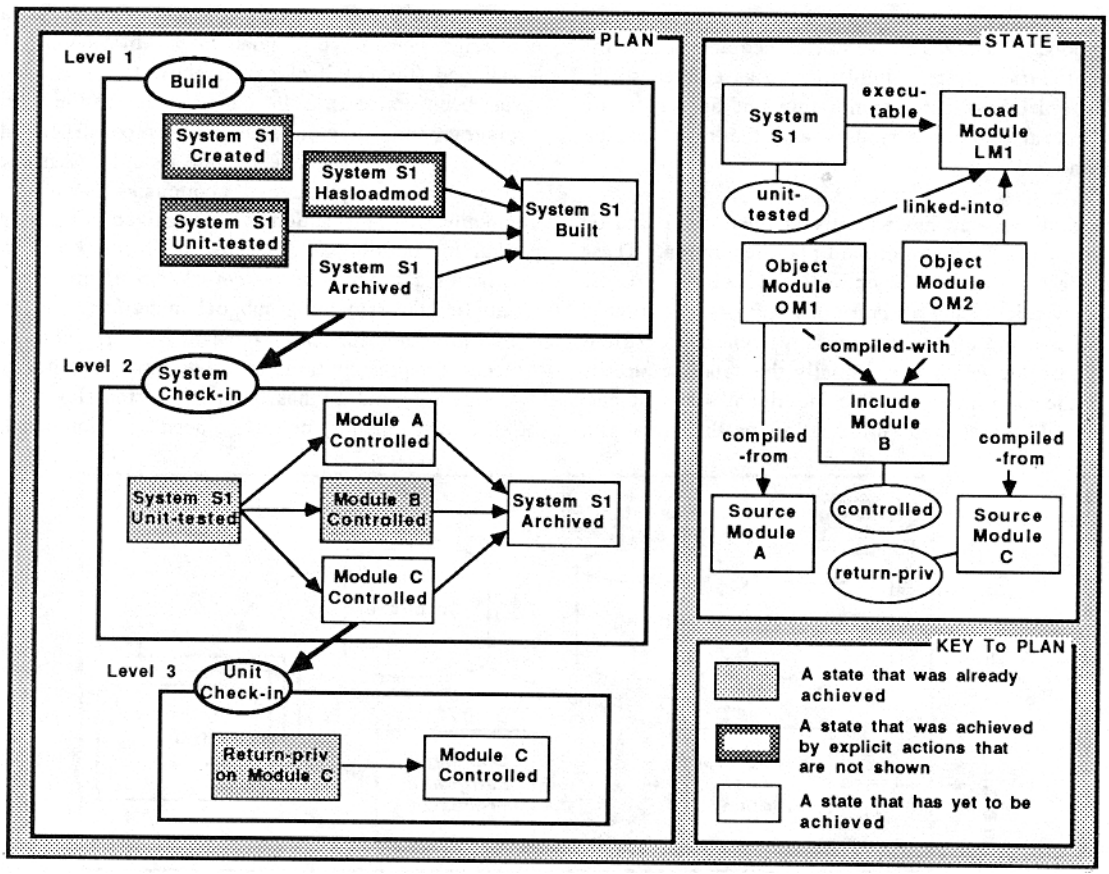

Figure 6: A Plan in Progress

Planning can be invoked on any level goal within an existing plan, or on new goals. Plan recognition sets the context for planning in two ways. First, it provides additional state information not otherwise available (e.g., the effects of *release*, or any effect in a complex operator not logically implied by the conjunction of the subgoals). Second, when the goal is part of a recognized plan in progress, some planning choices, such as parameter bindings, may already be decided.

3. Achieving Deeper Process Understanding

One obstacle to deeper process understanding is that some useful information about the state of the world cannot be directly observed from the actions performed; nor can it be computed with certainty from observable data. Consider the issue of the "releasability" of a given system version. Customers generally expect (bug-free) releases with successively greater functionality. The *release* operator of Figure 2 ignores these considerations—it is too *permissive*. For example, it allows releases in random order of functionality. Adding a requirement that a new release must have been developed after the current release gives an operator that is too *rigid*. A development time-stamp is not a perfect predictor of functionality, although generally versions developed later have more function. Also, there could be other exceptions, such as re-releasing the previous release when the current release is found to have a serious bug.

Although a "decision procedure" to determine releasability with certainty seems unattainable, it is possible to identify some rules for making plausible assumptions about releasability, based on what is typically the case. These assumptions would then be the basis for determining the *credibility* of a particular *release* action; actions below a certain credibility threshold could be challenged, not as actual errors, but as possible errors. If the programmer confirms the action, the assumptions would need to be revised to be consistent with the programmer's judgment.

Reasoning that involves making and revising plausible assumptions in the absence of complete information is called *non-monotonic reasoning* (NMR) [9]; one form of NMR is a *truth maintenance system* (TMS) [7]. In the next two sections, we show how deeper process understanding can be achieved using a TMS.

3.1 Formalizing Additional Knowledge

A TMS uses a multi-valued logic approach to NMR. It maintains a network of *nodes*, each of which can be labelled IN or OUT. Separate nodes are used for a predicate and its negation. If the node for a predicate is IN and the node for its negation is OUT, the predicate is *true*; if the node is OUT and the negation is IN, the predicate is *false*. If both are OUT, the truth value is *unknown*; if both are IN, there is a *contradiction*.

Justifications capture the relationships between the nodes, correlating a set of *support* nodes and a set of *exception* nodes with a *conclusion* node. A justification of the form A EXCEPT B —> C means if A is IN and B is OUT, then C is IN. The exception node B represents the non-monotonic content of the justification; a monotonic justification (standard logical implication) has an empty list of exceptions. In order for a node to be IN, it must have at least one *valid* justification; a justification is valid if all its support nodes are IN and all its exception nodes are OUT. A *premise* justification has empty support and exception lists; it is always valid.

Process knowledge about releasability is given in justification form in Figure 7 (left column). Rule J1 covers the common situation: a non-buggy version developed after the current release is considered releasable. Rule J2 covers a re-release scenario; if the current release is buggy, the

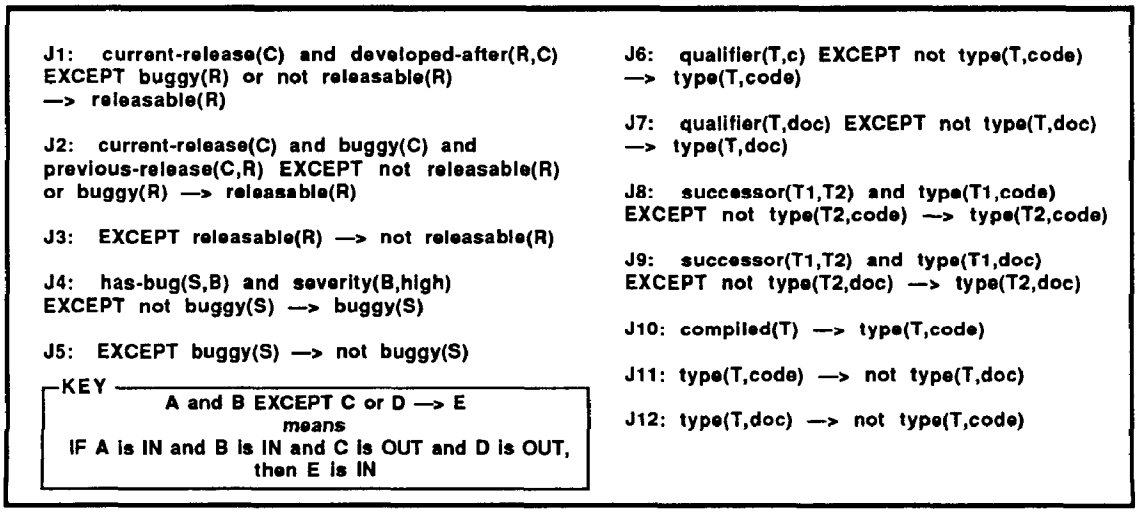

Figure 7: TMS Justifications

previous release is considered releasable unless it is buggy. Rule J3 provides that if rules J1 and J2 don't apply, a version is considered not releasable.* Rules J4 and J5 give a (simplistic) definition of the predicate buggy, based on the descriptions of explicitly reported bugs in a bug database. An additional set of justifications (Figure 7, right column) shows how the *type* of a text file can be deduced (based on either the qualifier on its file name or the type of its predecessor). This set contains monotonic justifications; for example, if the file has compiled successfully, then it is definitely code.

These justifications provide a way to compute the truth values of new predicates (*releasable*, *buggy*, and *type*), thus enlarging the state of the world beyond the original, *core* state. When the justifications are instantiated, and the truth values of core predicates entered (with premise justifications), the truth maintenance process will label the nodes, giving a read-out on the truth values of the new predicates. When the core state changes as a result of an action, the nodes will be relabelled and the new predicates may change. The TMS will also determine whether a node is *certain* or *by-assumption*. Nodes belonging to the core state are always certain. Other nodes are certain if they are the conclusions of monotonic justifications, all of whose support nodes are certain. Remaining nodes are by-assumption; for nodes that are IN by-assumption, one or more non-monotonic rules were needed to justify them. (The labels of nodes that are certain cannot be changed when assumptions are being revised.)

* A justification of the form EXCEPT A —> ¬A is only made valid *after* it has been determined that there are no valid justifications for A. This is a special case of a more general feature that allows justifications to be prioritized according to their predictive accuracy. Priorities ensure that stronger justifications are not invalidated by weaker ones, as described in [13].

A TMS labelling is shown in Figure 8, where Sys2 is the current release, Sys1 the previous release, Sys2 was developed after Sys1, and Sys3 was developed after Sys2. In the absence of any reported, high-severity bugs, none of these systems is buggy (J5 valid, J4 not valid in each case). Sys1 is not releasable—neither J1 nor J2 are valid, but J3 is valid. Sys3 is releasable by J1. Sys2 was at one time releasable, but this is no longer the case.

Since the justifications determine the truth status of the new predicates, *releasable* can be used where needed in operator definitions. In particular, *releasable(System)* can now be added as a constraint in the *release* operator of Figure 2.

3.2 Credibility and Revision of Assumptions

In the enlarged world state that is now accessible, predicates will evaluate to one of five truth values: true with-certainty, true by-assumption, unknown, false by-assumption, or false with-certainty. These correspond to five credibility classes, informally described as certainly OK, credible, can't tell, not credible, and certainly not OK. As the preconditions and constraints of an action are checked along the path from the action to a top-level goal, the credibility rating can be accumulated.

Credibility can be used in two ways: to select among competing interpretations for an action and to flag potential errors in actions. For example, if a file is assumed to contain code, we expect to see it used in actions on code, not actions on documentation. When there is an action involving this file that could be either part of preparing code or part of preparing documentation, the interpretation for preparing code will have a higher credibility, and will be preferred. Potential errors flagged for the programmer's attention correspond to interpretations with a "not credible" rating.

The internal structure of a TMS is designed for revising assumptions, not just reporting assumptions. When

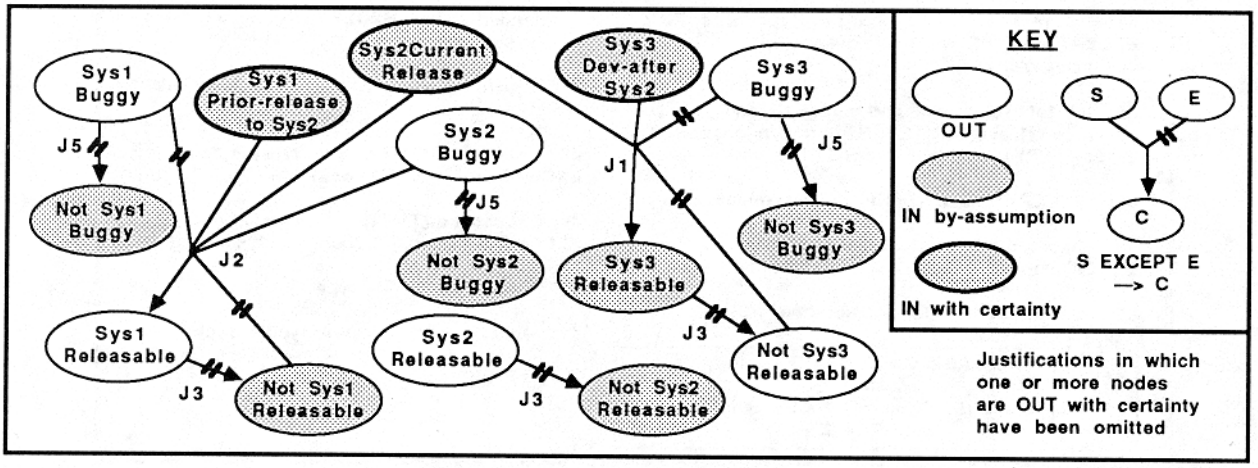

Figure 8: A TMS Labelling

proceeding with an interpretation in which one or more predicates evaluated to unknown or false by-assumption, it is necessary to make the world state consistent with the requirements of the action. This can be trivially accomplished by adopting the desired assumptions; but then, clues that other assumptions are wrong will be ignored. Since the justifications provide the accepted rationale for various assumptions, it is better to find a rationale for the desired assumption than to adopt it outright. This will "integrate" the programmer's judgment into the current set of assumptions.

Consider an action (re-)releasing Sys1 in the state diagrammed in Figure 8. Since *releasable(Sys1)* is false by-assumption, this action will be challenged. If the programmer confirms this action, then *releasable(Sys1)* must be made true. Only two rules, J1 and J2, justify *releasable*. J1 cannot be made valid—it has a support node *(developed-after(sys1,sys2))* that is OUT with certainty. J2 can be made valid by making *buggy(Sys2)* IN. This has to be done by supporting *buggy(Sys2)* directly, since no revision of assumptions can make J4 valid. This change and the new labels are shown in Figure 9. The algorithm for revising assumptions [13] is similar to dependency-directed backtracking [7].

This example shows how the process assistant uses the justifications to make independent assessments (originally to conclude that Sys1 is not releasable), and yet accede to a different decision and supply a rationale for that decision (that Sys1 is releasable *because Sys2 must be buggy*).

4. Status and Future Work

We have built a testbed for experimentation with the GRAPPLE plan-based process assistant. Two versions of the plan recognizer have been implemented. The TMS and related facilities for deeper process understanding are implemented in a Prolog version of the plan recognizer. GRAPPLE is not tied to a particular software environment; rather, it accepts command streams transcribed from actual terminal sessions or fabricated for experimental purposes. We have studied actual session transcripts to develop a better understanding of the content of process definitions, but have not yet tested GRAPPLE in a real setting.

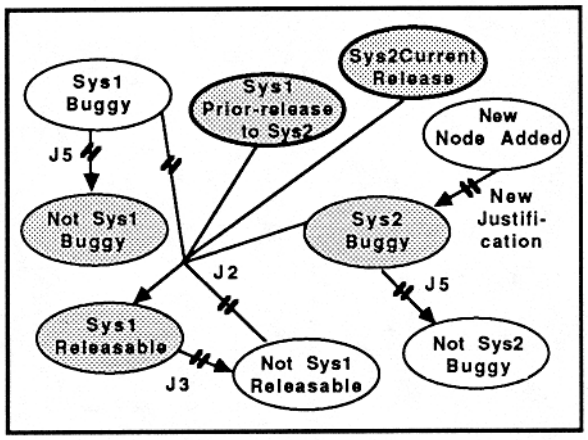

Figure 9: Revised Assumptions

There are several dimensions along which plan-based process support can be extended. We have already developed (and are currently implementing) a method of using metaplans to represent error recovery strategies and other types of knowledge of exceptional situations [12]; this is particularly important in light of the "trial and error" character of software development. We also want to explore the feasibility of applying intelligent assistance to the commands within an interactive tool (e.g., a syntax-directed editor). This would extend support to lower-level process activities and capitalize on synergy between levels, as the external context affects what is done within the tool, and vice versa. The most exciting extension involves the use of multi-agent planning techniques to represent project-level coordination, cooperation, and negotiation among programmers.

5. References

[1] Balzer, R.M. "Living in the Next Generation of Operating System". *IEEE Software*, 4:6 (November 1987), 77-85.

[2] Balzer, R.M.; Cheatham, T.E.; and Green, C. "Software Technology in the 1990's: Using a New Paradigm". *IEEE Computer*, (November 1983), 39-45.

[3] Bernstein, P.A. "Database System Support for Software Engineering". *Ninth International Conference on Software Engineering*, (March 1987), 166-178.

[4] Bisiani, R.; Lecouat, F.; and Ambriola, V. "A Planner for the Automation of Programming Environment Tasks". *Twenty-first International Hawaii Conference on System Sciences*, (January 1988).

[5] Chen, P.P. "The Entity-relationship Model: Toward A Unified View of Data". *ACM Transactions on Database Systems*, 1:1 (March 1976), 9-36.

[6] Croft, W.B. and Lefkowitz, L.S. "Knowledge-based Support of Cooperative Work". *Twenty-first International Hawaii Conference on System Sciences*, (January 1988), 312-318.

[7] Doyle, J. "A Truth Maintenance System". *Artificial Intelligence*, 12 (1980), 231-272.

[8] Fickas, S.F. "Automating the Transformational Development of Software". *IEEE Transactions on Software Engineering*, SE-11:11 (November 1985), 1268-1277.

[9] Genesereth, M.R. and Nilsson, N.J. *Logical Foundations of Artificial Intelligence*. Palo Alto, California: Morgan Kaufmann, 1987.

[10] Huff, K.E. "A Database Model for Effective Configuration Management". *Proceedings of Fifth International Conference on Software Engineering*, (March 1981), 54-61.

[11] Huff, K.E. and Lesser, V.R. "Knowledge-based Command Understanding", Technical Report 82-6, Department of Computer and Information Science, University of Massachusetts, Amherst, 1982.

[12] Huff, K.E. and Lesser, V.R. "Metaplans That Dynamically Transform Plans", Technical Report 87-10, Department of Computer and Information Science, University of Massachusetts, Amherst, 1987.

[13] Huff, K.E. and Lesser, V.R. "Plan Recognition in Open Worlds", Technical Report 88-18, Department of Computer and Information Science, University of Massachusetts, Amherst, 1988.

[14] Johnson, W. and Soloway, E. "PROUST: Knowledge-Based Program Understanding". *IEEE Transactions on Software Engineering*, 11:3 (March 1985), 267-275.

[15] Kaiser, G.E. and Feiler, P.H. "An Architecture for Intelligent Assistance in Software Development", *Proceedings of the Ninth International Conference on Software Engineering*, (1987), 180-188.

[16] Leblang, D.B. and Chase, R.P. "Computer-aided Software Engineering in a Distributed Workstation Environment". *Proceedings of SIGSOFT/ SIGPLAN Symposium on Practical Development Environments*, (1984), 104-112.

[17] Luria, M. "Goal Conflict Concerns". *Proceedings of IJCAI*, (August 1987).

[18] Rich, C. and Shrobe, H.E. "Initial Report on a Lisp Programmer's Apprentice". *IEEE Transactions on Software Engineering*, SE-4 (November 1978).

[19] Osterweil, L. "Software Processes are Software Too." *Proceedings of the Ninth International Conference on Software Engineering*, (1987), 2-13.

[20] Penedo, M.H. and Stuckle, E.D. "PMDB—A Project Master Database for Software Engineering Environments". *Proceedings of the Eighth International Conference on Software Engineering*, (1985), 150-157.

[21] Ramamoorthy, C.V.; Usuda,Y.; Tsai, W.T.; and Prakash, A. "GENESIS: An Integrated Environment for Supporting Development and Evolution of Software", *IEEE Ninth International Computer Software and Applications Conference*, (October 1985), 472-479.

[22] Sacerdoti, E.D. *A Structure for Plans and Behavior*. New York: Elsevier-North Holland, 1977.

[23] Tate, A. "Project Planning Using a Hierarchical Non-linear Planner". Department of Artificial Intelligence Report 25, Edinburgh University, 1976.

[24] Tichy, W.F. "RCS—A System for Version Control". *Software Practice and Experience*, 15:7 (July 1985), 637-654.

[25] Waters, R.C. "The Programmer's Apprentice: A Session with KBEmacs". *IEEE Transactions on Software Engineering*, SE-11:11 (November 1985), 1296-1320.

[26] Wile, D.S. and Allard, D.G. "Worlds: an Organizing Structure for Object-Bases". *Proceedings of Second SIGSOFT/SIGPLAN Symposium on Practical Development Environments*, (1986), 16-26.

[27] Wilensky, R.; Arens, Y.; and Chin, D. "Talking to UNIX in English: An Overview of UC". *CACM*, 27:6 (June 1984), 574-593.

[28] Wilkins, D.E. "Domain-Independent Planning: Representation and Plan Generation". *Artificial Intelligence*, 22 (1984), 269-301.

[29] Williams, L. "Software Process Modeling: A Behavioral Approach". *Proceedings of the Tenth International Conference on Software Engineering*, (1988), 174-186.