Meta-plans That Dynamically Transform Plans

Karen E. Huff and Victor R. Lesser

COINS Technical Report 87-10
November 1987

Computer and Information Science Department
University of Massachusetts
Amherst, MA. 01003

ABSTRACT: We present a general approach to augmenting the representational power of hierarchical plan formalisms. In complex domains, this additional power is needed to capture knowledge dealing with such issues as special cases of operators and strategies for recovering when operators fail. We describe limitations inherent in operator definitions and show that they can be overcome by expressing domain knowledge as transformations on plans. These transformations *reformulate* a (partially developed) plan, tailoring it rather than contributing directly to its completion. Since transformations are operations on a world state representing the plan network, they can be formalized as meta-operators and synthesized into meta-plans. The advantage of this approach is expressive completeness as compared to introducing special-case constructs into the operator definition language.

1.0 Introduction

Planning and plan recognition systems derive their power from having general-purpose algorithms that bring domain-specific knowledge to bear; their power is directly related to the extent of the domain knowledge that can be supplied. A limiting factor on providing this knowledge is the representational adequacy of formalisms for defining domain operators. Particularly in the case of complex domains, there are problems in capturing relevant domain knowledge such as how to recover when an operator fails or when to use special case operators. The challenge is to provide this knowledge in a way that is tractable both to the planning algorithms and to the author of the domain operators.

We encountered these issues as part of designing and developing an intelligent assistant, named GRAPPLE, based on a plan recognition and planning paradigm [1,2,3,5,6]. GRAPPLE provides passive assistance by monitoring user actions, performing plan recognition to detect and correct errors in the user's plan; it provides active assistance by cooperative generation of plans to meet user-stated goals. The test domain for GRAPPLE is software development [8], specifically programming performed in a traditional language such as C. A partial library of (simplified) operators for this domain is sketched in Figure 1. These operators have the usual clauses stating goals, preconditions, effects (generally, a superset of the goals), and constraints. Primitive operators correspond to the atomic actions in the domain; complex operators have subgoals that decompose the operator into simpler steps.

1.1 Limits on Representational Power

Hierarchical plan systems, based on NOAH [12] and NONLIN [14], are particularly appropriate for handling the operator libraries of complex domains. The use of non-primitive operators allows activities to be defined at multiple levels of abstraction, with more or less detail as appropriate; this provides an orderly approach to covering all domain activities. The modularity of operator libraries facilitates several aspects of working with large numbers of operators. Following the principles of information-hiding[11], certain

```
OPERATOR build
GOAL:        status(?system,built)
PRECOND:     true
SUBGOALS:    created(?system,?baseline)
             status(?system,compiled)
             status(?system,linked)
             status(?system,tested)
EFFECTS:     SET status(?system,built)
```

```
OPERATOR release
GOAL:        status(?system,released)
PRECOND:     status(?system,built)
CONSTR:      current-release(?c)
EFFECTS:     SET status(?system,released)
             DELETE current-release(?c)
             SET current-release(?system)
             SET customer-release(?system)
```

```
OPERATOR test
GOAL:        status(?system,tested)
PRECOND:     true
SUBGOALS:    unit-tests(?system,ready)
             unit-tests(?system,run)
EFFECTS:     SET status(?system,tested)
```

```
OPERATOR link
GOAL:        status(?system,linked)
PRECOND:     status(?system,compiled)
EFFECTS:     NEW load-module ?lm
             SET status(?system,linked)
             ADD lm-for(?system,?lm)
             SET time(?system,?time)
```

<u>Relationships</u>

Build satisfies precondition of Release
Link satisfies subgoal of Build
Test satisfies subgoal of Build

Figure 1:
A Partial Library of
Simplified Software
Process Operators

details can be encapsulated in one or a few operators; later, if those details must change, the effects are local, not global. Great flexibility is obtained if the decomposition of operators is always stated in terms of states to be achieved, not directly in terms of the operators that achieve those states; the operators of Figure 1 follow this style. The advantage is that when a new operator, satisfying a state required in other operators, is added to the library, the existing operators do not have to be modified to mention the new operator. This capitalizes on the fact that operators are potentially applicable in *any* context where their goals match the preconditions or subgoals of other operators. In general, operators can be written without knowledge of the other operators — either those operators that achieve the same or similar states, or those operators that require particular states in their decomposition.

In complex domains, cases arise where appropriate expressive power is lacking in the operator formalism; attempts to describe certain operators accurately can jeopardize the library modularity, or fail outright. We discuss several such problems, using examples of special cases of operators and failure recovery strategies taken from the software development domain, in the remainder of this section.

Adding special case operators to a library may require that preconditions or subgoals of existing operators be rewritten. For example, when testing a system that is intended to fix certain bugs, the programmer should run the official testcases associated with those bugs, in addition to those testcases that would otherwise be selected. One solution (that keeps testing considerations local to the set of testing operators) is to write a separate operator covering all testing needed when bugs are being fixed; its precondition restricts its applicability to systems intended to fix bugs. Now there are two operators for testing that are intended to be mutually exclusive. Therefore, the normal operator for testing must specify in its precondition that it is not applicable to cases where bugs are being fixed[1].

To accommodate other types of special cases, existing operators may have to be rewritten in artificial ways. Consider testing a system that is about to be released to a customer; such testing should include running the testcases in the regression test suite[2] (again, in additional to normally selected testcases). The precondition for this special operator concerns the existence of a goal to release the system; while the goal formula is expressible using domain predicates, the fact that a goal with this formula is currently instantiated is not expressible in domain terms. The only recourse is to write separate operators with artificially different goals. Then, operators (like *build*) that have testing subgoals will be affected, defeating the aim of encapsulating testing considerations within the testing operators. Thus, the designer of the operators must produce not only a normal *test* operator and a *test-for-release* operator, but also a normal *build* operator and a *build-for-release* operator, to ensure that the right type of testing is performed in all cases.

Expressing special cases with this brute force approach, already attended with disadvantages, breaks down entirely when multiple special cases affect a single operator; the combinatorics are intolerable from the designer's perspective. Special cases are not

---

[1] One could institute a fixed preference strategy to select the operator with the most specific precondition that can be satisfied. However, in general this is overly restrictive — it would prevent a car buyer from financing his purchase by selling stock to raise funds because taking out a car loan is the most narrowly applicable operator.

[2] In software engineering, regression testing is performed to ensure that bugs have not been introduced into functions that were previously shown to work correctly.

guaranteed to be simply additive with respect to the normal case. At worst, separate operators must be provided for all *combinations* of special factors.

In dealing with recovery from operator failure, there are problems both in connecting the right recovery operator with a failure situation, and in simply expressing the recovery strategy itself. Sometimes special operators are used for failure recovery, and only for failure recovery; for example, one of the actions for dealing with a compilation failure due to bugs in the compiler is to report the compiler bug. *Report-tool-bug* can be written as an operator, but how will such an operator get instantiated? Missing are the constructs indicating what goals (and therefore what operators) should be instantiated when a failure occurs. At other times, the recovery strategy may involve executing some normal operator in a special way. If the *build* operator fails because the *system* being built is faulty (as would be the case if the linker detected programming errors), then one recovery strategy is to restart the *build* process using the faulty *system* as the *baseline* from which to edit. Expressing such a strategy requires access to the variable bindings of operator instances; again, this is beyond the scope of domain predicates.

1.2 Extending Representational Power

These problems have previously been tackled separately on a case-by-case basis, introducing special operator-language constructs covering selected cases and providing domain-independent strategies that can be tailored in fixed ways. McDermott's policies [10] represent one approach to the issue of matching special case operators to the appropriate ckrcumstances. These policies derive their power from the fact that the NASL language allowed the writer to use plan-oriented constructs, such as *<policy> IMPLIES (TO-DO <task> <operator>)* or *<policy> IMPLIES (RULE-OUT <operator>)*, in addition to domain-oriented constructs. Recovery from failure of operators is addressed in SIPE [16]. There, a special error recovery language is defined; domain-independent strategies, such as reactivating a goal (RETRY in SIPE terms), are parameterized to allow deployment in operator-specific ways.

In this paper we describe a *single* formalism that extends the representational power of hierarchical planning paradigms. Our approach is based on abandoning the notion of predefining *all* operators; instead, we define transformations to be applied to *instances* of operators within a plan to create variations in response to special circumstances. Transformations are operations on some world state; in this case, the world state *is* the plan network. Therefore such transformations can be formalized as meta-operators and synthesized into meta-plans. This approach has the desired generality; it also adds to the role of meta-plans, which have been previously use to implement control strategies [13] and to capture domain-independent knowledge [15].

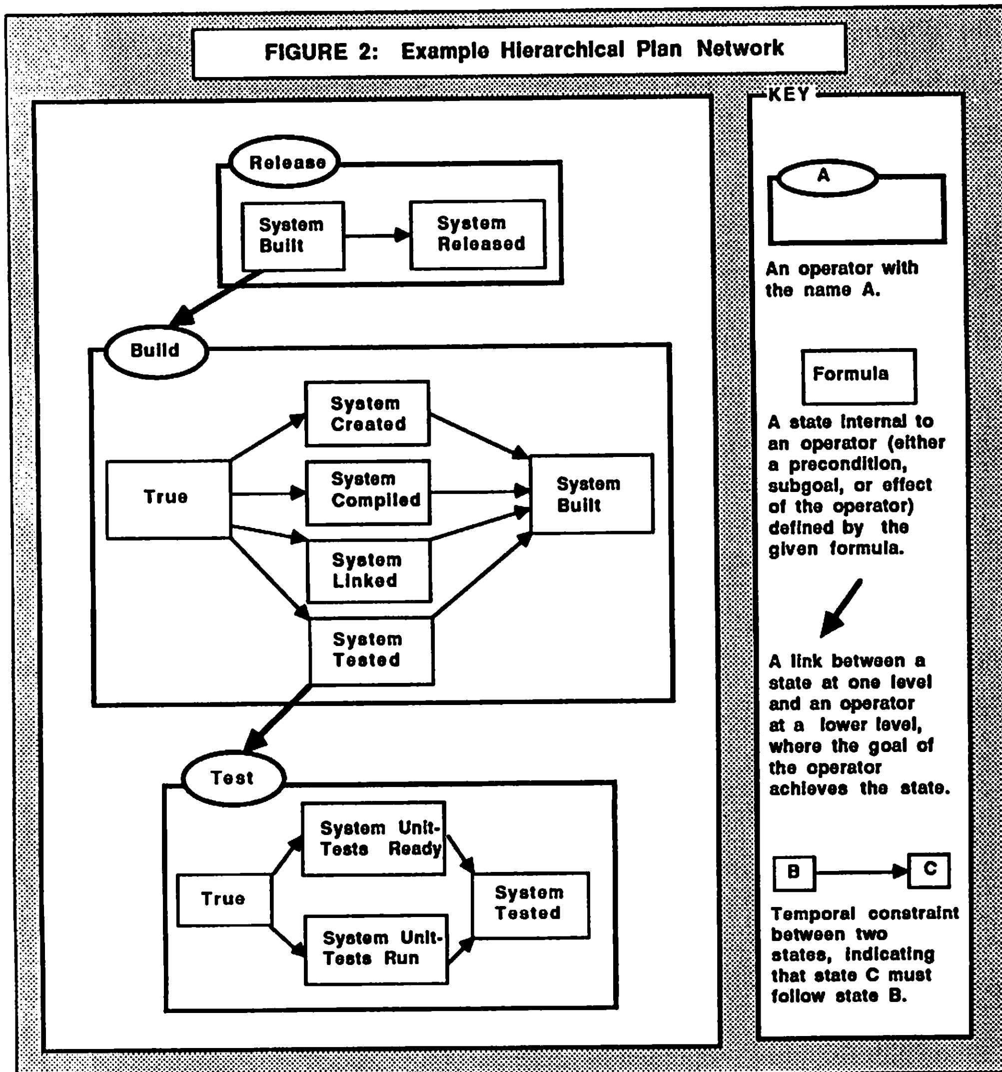
The primary advantage of this method is expressive generality, as compared with a collection of *ad hoc* operator language extensions. Any aspect of an operator definition (such as preconditions, subgoals, constraints, or effects), as well as any aspect of an operator instance (such as bindings of variables or ancestor operator instances) can be accessed or modified. The transformational approach also addresses some practical problems associated with developing and maintaining a complete library of operators. Because knowledge of exceptions is partitioned from knowledge of normal cases, the two issues can be tackled separately. The process of writing operators is further improved because multiple transformations can apply to a single operator, thereby preventing combinatorial explosion in numbers of operators.

In the remainder of the paper, we introduce the transformational approach with some specific examples from the software development domain. Then, we discuss how the transformations are formalized and expressed as meta-operators; both the state description and required operator constructs are covered. Finally, we present status and conclusions.

2.0 Transformations on Plans

2.1 Plans As Networks

The basic data structure of a planner or plan recognizer is a hierarchical plan network as first developed in NOAH[12]. An example of such a network (using some of the plans of Figure 1) is given in Figure 2. There, a vertical slice through a network covering three hierarchical levels is shown, with the highest level at the top of the figure. Downward arrows between levels connect desired states with operators chosen to achieve them. Such

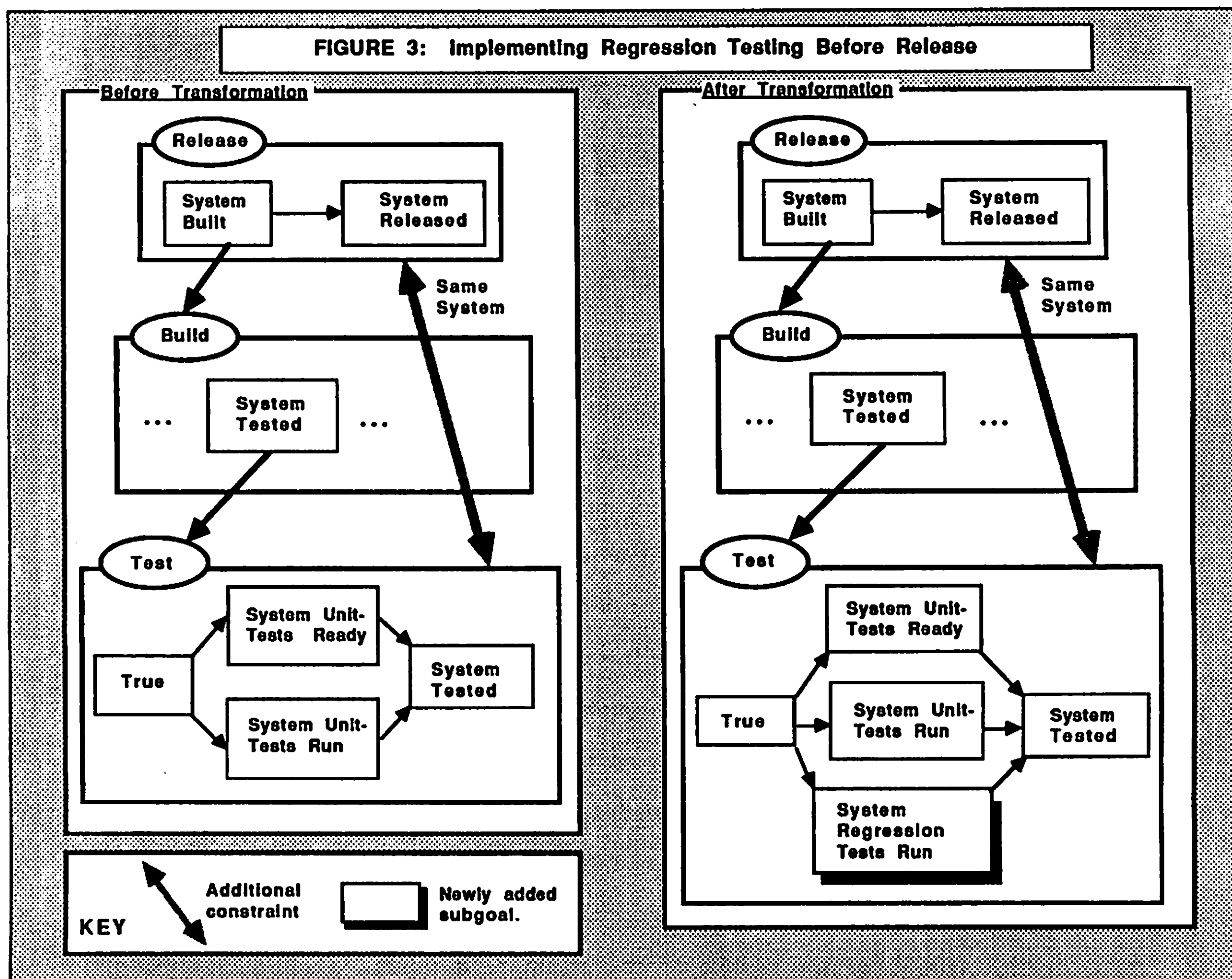


instantiated operators consist of additional states describing the internals of the operator: preconditions, subgoals, and effects. Arrows within levels show how the achievement of certain states is partially ordered with respect to time (some orderings are specified by the operator definitions and other orderings are imposed to resolve interactions). Orderings are propagated from level to level, but have been omitted to simplify the figure.

Both planning and plan recognition involve building a complete plan network. This is done by actions such as choosing operators to achieve states, instantiating these operators, and resolving conflicts between newly revealed states and existing states. Each such action can be thought of as taking the plan network one step closer to completeness. In contrast, a transformation will reformulate the current state of the network, with the effect of changing the solution set that will be pursued to complete the network. Such a reformulation is necessary either because the existing state of the network does not accurately reflect special circumstances or because other actions have reached a dead end (for example, a plan failure has occurred). Reformulating the network represents a permanently- instantiated objective of the planner/recognizer. Thus, the execution of (top-level) transformations will be data-driven: they will be applied whenever the current state of the network indicates an opportunity to do so.

2.2 Example: A Special Case

Software development is an example of a domain where a large part of the knowledge about how actions are carried out is concerned with special cases. Consider a transformation that implements the requirement to do regression testing before a release. When expressed precisely, the transformation affects an instance of *test* occurring as part of the expansion of *release*. To be entirely safe, one additional restriction should be given: that the *system* being tested is the same as the *system* being released. This will allow other testing instances to occur in the same expansion (such as running a testcase to help decide what editing changes are needed), while ensuring that regression testing is required on the right one. Expressing this condition requires access to the dynamic correspondence between the variable names used in the two operators. The BEFORE case of Figure 3 shows one situation in which this transformation is applicable.



Assuming the *test* operator of Figure 1, the effect of the transformation is to add an additional subgoal to run the regression test cases. The formula defining the new subgoal is supplied explicitly in the transformation -- it need not have appeared previously in the plan network. Only the one operator instance is modified; the basic operator definition for *test* is unchanged. The results of applying this transformation are shown as the AFTER case in Figure 3.

2.3 Example: Failure Recovery

Software development is also a domain where there are many causes of failure, including system problems, tool problems and programmer error. In particular, given that much work is carried out on a trial-and-error basis, failures due to programmer error are to be

expected frequently. Consider the case of the *link* operator failing to produce a load module because errors made by the programmer were detected. In fact, decisions about recovery from this failure are not made at the local level of the *link* operator; a *link* operator failure implies that the parent operator has failed, and the appropriate recovery strategy is dictated by what that parent operator is.

The parent operator will usually be the *build* operator. If the *build* operator has failed and the *system* that was built is faulty, one recovery scenario is to go through the whole *build* process again; but, instead of starting from the same *baseline* used in the original *build* instance, this new *build* instance will start with the faulty *system* as the *baseline.* That is, the new *build* instantiation will use as the binding of its *baseline* variable the binding of the *system* variable from the failed *build* instantiation.

These strategies can be expressed in two separate transformations. The first transformation applies to instances of *link* that have failed; its effect is simply to mark the parent operator instance as failed. The second transformation applies to instances of *build* that have failed; its effect is to create a new instantiation of the *build* operator, and to fix the binding of the *baseline* variable in that instantiation to be equal to the binding of the *system* variable from the "superseded" instantiation of *build.* This is shown in Figure 4.

2.4 Other Examples

The software development domain is particularly rich in examples that demonstrate the generality of the transformational approach. Some additional generic uses of transformations, beyond representing special cases and straightforward failure recovery strategies, are these:

- Transformations can be used to maintain desirable domain states in a flexible manner (McDermott used policies this way[10]). That is, whenever an undesirable state obtains, a goal can be posted to reestablish the desired state. This is more forgiving than preventing the undesired state absolutely. As an example, programmers generally follow a set of rules about how files are allocated to directories. However, in the heat of activity, a file may be created

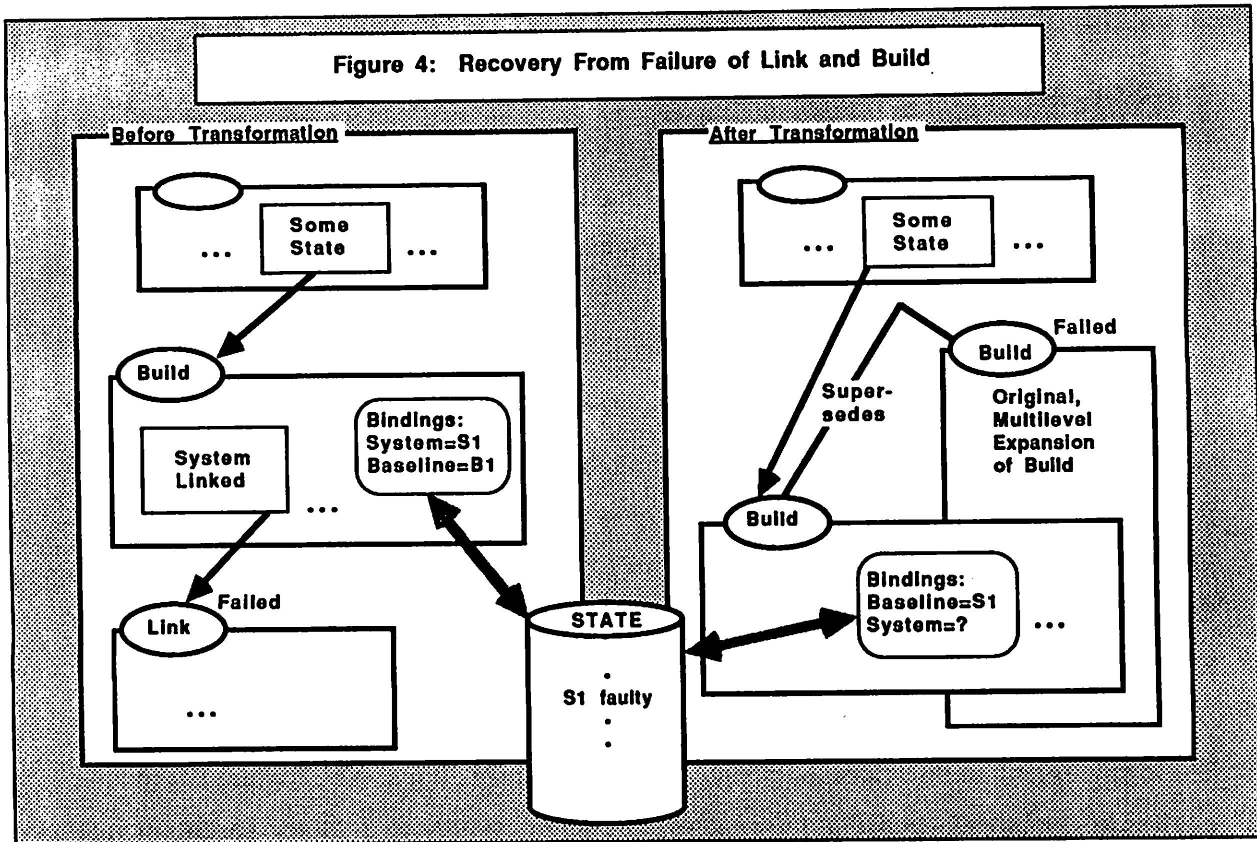


Figure 4: Recovery From Failure of Link and Build

in the "wrong" directory. A transformation could trigger on this and instantiate a goal to move the file to the proper directory.

- One method of handling an adverse interaction between operators for achieving parallel goals is to place temporal constraints on the order in which the operators are executed. This method has the advantage of being domain-independent, but there may be domain-dependent techniques as well. These can be captured in transformations whose preconditions are that adverse interactions between two planned actions have been detected. In the software domain, the operator that copies the contents of one file into another can be inserted into a plan to correct some types of interactions. However, some explicit clue, such as a transformation, is needed to ensure that copy is considered for handling interactions.

- Transformations can be used to apply shortcuts in just those situations where the shortcut is safe. A shortcut amounts to substituting one goal which is "easier" to achieve for another which is "harder" to achieve. The safety of the shortcut may involve the context in which the goal is instantiated, so the meta-level constructs of the transformation are doubly necessary. In the software domain, if editing a source module consists of cosmetic changes only, then an alternative to recompilation is simply to acquire (and place in the appropriate

directory) the object module of the previous version (assuming no include modules were also changed). However, it is bad practice to do this on a release to a customer. Only by expressing this in a transformation can we ensure that good practice is followed.

- In some cases when operators fail, the recovery strategy may involve rephrasing the goal in order to proceed with the overall plan. In these cases, a failure indicates that the goal itself (in all its detail) cannot be accomplished; but there may be a related goal that will suffice. A transformation, whose precondition is that an operator failed to achieve its goal and whose effects are to substitute a different goal for the original goal, can express this strategy. A software example arises if the compiler blows up when directed to compile at its highest optimization level. A well-established strategy is to try again with optimization turned off. If this results in a successful compilation, the programmer will settle for a load module which is only partially optimized.

- Transformations can apply to a specific operator (such as the *test* operator examples given earlier), or to any operator having a particular characteristic, such as a specific goal or subgoal formula. They can also apply to arbitrary characteristics of operator definitions, such as any operator having a particular type of parameter or parameter binding. In a multi-user system, when the number of users logged-in is below a certain threshold, then commands will be submitted for foreground execution rather than to a background queue. Suppose all operators representing CPU-intensive commands are written with an explicit parameter set for background execution. Then, one transformation, applying to any operator utilizing the background queue, can handle foreground/background selection. This transformation saves the author of operators from writing an additional version of each CPU-intensive operator.

- The examples of transformations given so far are relatively simple, most often requiring one operation on the plan net. However, transformations can be arbitrarily complex. In the software domain, recovering from failed operators includes deleting extraneous files. One transformation could identify certain files created by operators in the expansion of a failed operator and instantiate goals to delete them. This transformation applies one change (instantiate a goal with specific variable bindings) many times (for each selected file).

- Another complex transformation in the software domain applies to the conservative editing style of frequently saving a snapshot of the file being edited; here the intermediate snapshots must eventually be deleted. This is a complex transformation involving two separate (but related) changes in the plan net: one change instantiates goals to save snapshots, and the other change instantiates goals to delete all but the desired version.

- A final use of transformations is to allow sharing of a generic operator library among several applications, where each application has special requirements. In the software domain, several projects can share a generic operator library, if each expresses project-specific policies as transformations. Then, one project can require that a particular analysis tool be run before a customer release, without affecting whether other projects use the same tool in the same way.

3.0 Formalizing the Transformations

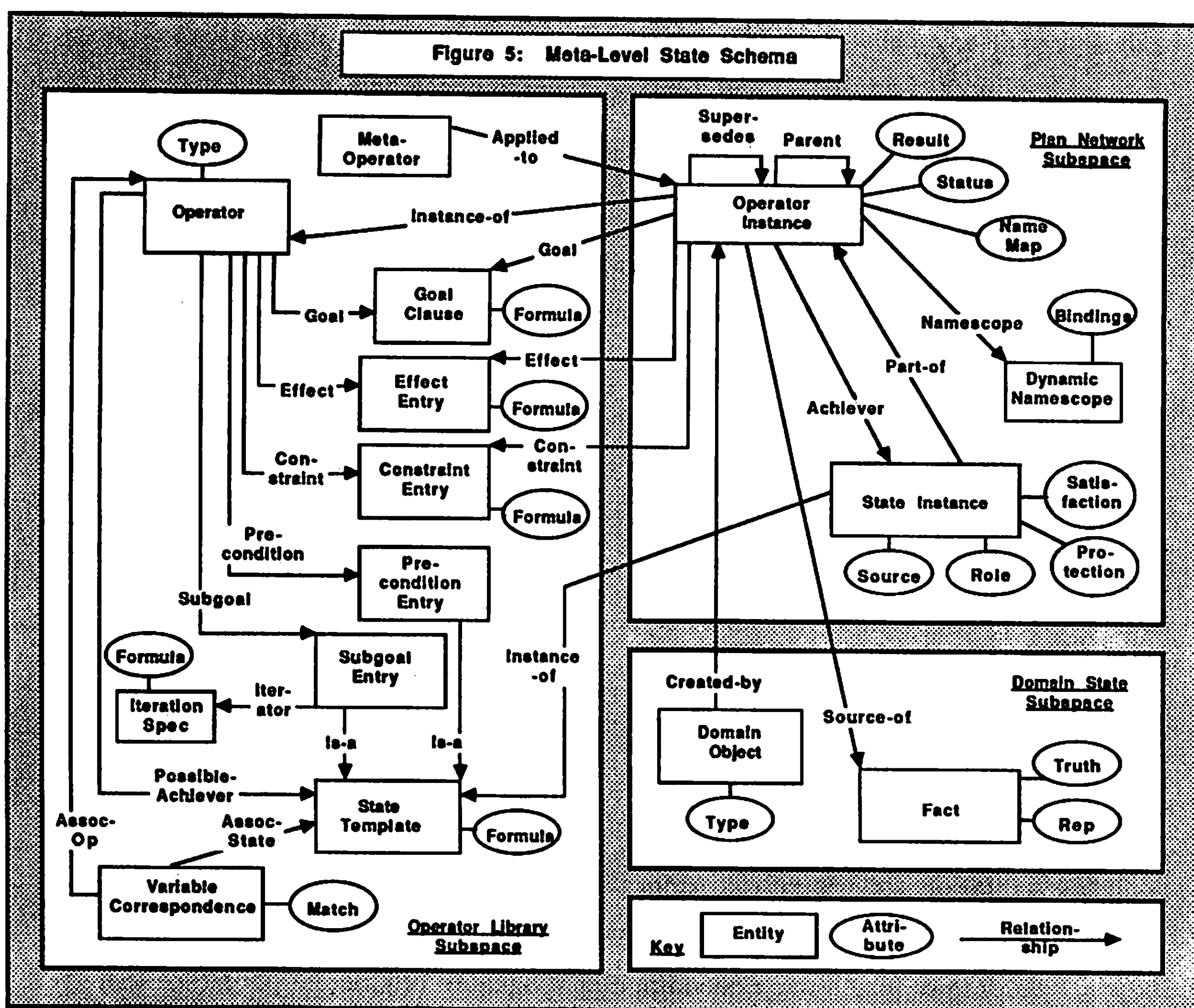
Because transformations are operations on the plan network, they can be formally represented as meta-plans, synthesized from meta-operators. Meta-plans are not a new idea. Procedurally-implemented meta-plans were introduced by Stefik[13] to pursue control issues in planning. Declarative meta-plans were defined by Wilensky[15] in order to share knowledge between a planner and a plan recognizer. Neither of these meta-plan systems was used to modify operator instances by adding new subgoals, changing constraints and so forth. Meta-plans that could modify steps or change parameter bindings were defined for a natural language dialog understanding system[9]. In these meta-plans, the modifications were meta-plan parameters which were bound from information in the utterances.

3.1 The Meta-level State Schema

The meta-level state schema covers most of the internal data structures used in planning. The entity-relationship (ER) model of data [4] provides a useful way to visualize such a complex state schema. In the ER model, there are entities which have attributes and participate in relationships with other entities. There is a straightforward translation between the ER model and predicate calculus, whereby relationships and attributes correspond to predicates. Semantic constraints can be expressed as axioms in predicate calculus. Other axioms can be used to define additional predicates and to define appropriate functions.

The ER diagram of the meta-level state schema is given in Figure 5. The objects and relationships shown are representative, not exhaustive. The schema describes operators as they are defined in the GRAPPLE formalism, and addresses the requirements of plan recognition (the context in which we are currently implementing the meta-plans).

The state schema for the meta-plans contains objects and predicates organized into three subspaces: operator library, plan network, and the domain state. The operator library subspace describes all components (preconditions, effects, etc.) of all operators, and their formulas and (static) variable names. The plan network subspace describes the hierarchy of operator instances: their dynamic status (started, completed, failed...), their internal



states and status (pending, achieved, protected...). Also associated with operator instances are the dynamic name scopes and variable binding information needed to evaluate operator formulas. The domain state represents the truth value of all domain predicates. Additional predicates cross subspace boundaries, relating operator instances back to their definitions, and operator instances to the domain predicates they affected.

The state schema is designed so that it represents a single choice from among the competing interpretations of a series of actions. Thus, if an operator can achieve the subgoals of two other operators, this will be represented using two states, each in a separate context. During recognition, there will usually be multiple contexts that are active; an important component of the recognizer comprises strategies for focusing among these alternatives, as well as for selecting the operations applied to an alternative. Thus we make the same distinction as in [13] between operations in planning space and the control strategies by which those operations are selected. The transformational meta-operators add to the number of operators subject to control decisions.

3.2 Meta-operator Constructs

Because the transformations are complex, expressing them as operators requires a language engineered for "real-world" use. Clearly, effects of operators must be able to create new objects (for example, a new subgoal instance). Some transformations (such as the one to delete extraneous files) require a facility for iterating subgoals: that is, for repeatedly achieving a subgoal formula over a set of bindings. Conveniently, all the needed facilities were already available in the formalism we designed to handle the complex domains anticipated for GRAPPLE [7]. No new facilities (except a keyword to distinguish between operators and meta-operators) were needed.

The transformation for regression testing before release is shown as a meta-operator (expressed in GRAPPLE notation) in Figure 6. This will be a top-level operator assuming that its goal does not match the precondition or subgoal of other meta-operators; therefore, it will be executed when its precondition becomes true. That will happen when there is an instance of *test* whose ancestor is an instance of *release* AND when the *system* variables of

Figure 6: Meta-operator for Regression Testing

```
(METAOPERATOR   regressions-before-release

(GOAL           applied-to(regressions-before-release,?test-op))

(PRECOND        (STATIC instance-of(?test-op,test) AND
                ancestor(?test-op,?rel-op) AND
                instance-of(?rel-op,release) AND
                same-dynamic-name(system,?rel-op,
                   system,?test-op) ) AND
                NOT applied-to(regressions-before-release,
                   ?test-op) ) )

(EFFECTS        (NEW state-instance ?regressions)
                (NEW subgoal-entry ?regr-subgoal)
                (NEW iteration-spec ?iterate-info)
                (ADD part-of(?regressions, ?test-op))
                (ADD instance-of(?regressions, ?regr-subgoal))
                (ADD iterator(?regr-subgoal, ?iterate-info))
                (ADD source(?regressions, metaplan))
                (ADD role(?regressions, subgoal))
                (ADD protection(?regressions, not-protected))
                (ADD satisfaction(?regressions, unknown))
                (ADD formula(?regr-subgoal,
                      tested-on(?system,?regr-case) ) )
                (ADD formula(?iterate-info,
                      in-regression-suite(?regr-case) ) )
                (ADD applied-to(regressions-before-release,
                     ?test-op) ) ) )
```

both operators correspond (e.g., are mapped to the same dynamic name). The precondition carries the keyword STATIC, meaning that no explicit attempt should be made to render it true. A state in which the transformation precondition is true is diagrammed in Figure 7.

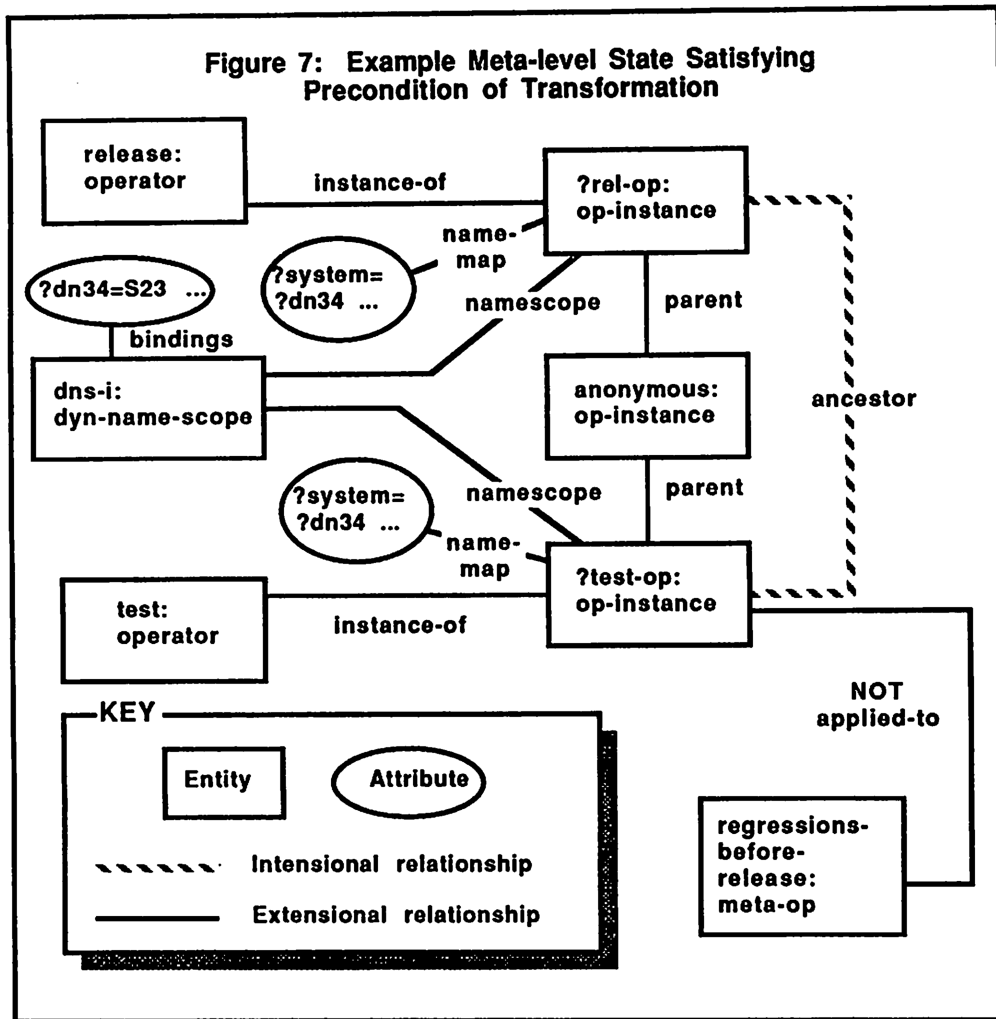


Figure 7: Example Meta-level State Satisfying Precondition of Transformation

Performing the transformation is simply a matter of realizing the effects of the meta-operator in this case, because there are no subgoals to be achieved. These effects are all directed at creating a new state instance as part of the *test* operator instance. The new state instance has a supporting subgoal entry that defines the state formula and, since it is an iterated subgoal, the iteration formula. Note that the meta-operator contains these formulas explicitly; they consist of domain predicates and variable names in the static name scope of the *test* operator definition. After the transformation has been executed, the precondition will no longer be true for this instance of *test*; thus, this transformation is not meant to be applied more than once to the same situation. The state shown in Figure 7 becomes the state diagrammed in Figure 8 after the transformation is applied; changes are highlighted.

Figure 8: Meta-level State after Transformation

The transformation exporting the failure of the *link* operator to its parent operator is shown in Figure 9, to show how a goal for failure recovery is expressed.

Figure 9: Meta-operator for Link Failure

```
(METAOPERATOR   propagate-link-failure
(GOAL           status(?link-op,failure-processed) )
(PRECOND        (STATIC status(?link-op, failed) AND
                instance-of(?link-op,link) AND
                query(?link-op, faulty(?system) ) ))
(CONSTRAINTS    (parent(?link-op,?parent-op))
(EFFECTS        (DELETE status(?parent-op,in-progress) )
                (DELETE status (?link-op,failed) )
                (ADD status(?parent-op,failed))
                (ADD status(?link-op,failure-processed)) ))
```

4.0 Status and Conclusion

Transformational meta-plans provide a powerful means of capturing and applying additional domain knowledge, which is key to achieving more powerful planning systems. Defining domain knowledge about special cases and failure recovery via meta-operators provides an expressively complete approach, which obviates the need for special-purpose language extensions. This approach also expands the role that meta-planning plays in a planning architecture. The resulting transformations represent a special class of operations on plan networks: operations that reformulate a network rather than solve it. As an additional benefit, this approach eases the writer's task of providing thorough coverage of domain actions. We are currently implementing the transformations as part of the GRAPPLE plan recognizer. This implementation uses Knowledge Craft™ and capitalizes on its facilities for context management, object schema, and integrated Prolog features. We are continuing to explore the role of deeper domain knowledge in planning systems, with particular emphasis on its relationship to control.

Knowledge Craft™ is a trademark of Carnegie Group Incorporated.

5.0 References

[1] Broverman, C.A., and W.B. Croft, "A Knowledge-based Approach to Data Management for Intelligent User Interfaces", *Proceedings of Conference for Very Large Databases*, 1985.

[2] Broverman, C.A., K.E. Huff, and V.R. Lesser, "The Role of Plan Recognition in Intelligent Interface Design, *Proceedings of Conference on Systems, Man and Cybernetics*, IEEE, 1986, pp. 863-868.

[3] Carver, N., V.R. Lesser and D. McCue, "Focusing in Plan Recognition", *Proceedings of AAAI*, 1984, pp. 42-48.

[4] Chen, P.P., "The Entity-relationship model: Toward A Unified View of Data," *ACM Transactions on Database Systems*, vol. 1, no. 1, March 1976, pp. 9-36.

[5] Croft, W.B., L. Lefkowitz, V.R. Lesser and K.E. Huff, "POISE: An Intelligent Interface for Profession-based Systems", Conference on Artificial Intelligence, Oakland, Michigan, 1983.

[6] Croft, W.B., and L.S. Lefkowitz, "Task Support in an Office System", *ACM Transactions on Office Information Systems*, vol. 2, 1984, pp. 197-212.

[7] Huff, K.E. and V.R. Lesser, "The GRAPPLE Plan Formalism", Technical Report 87-08, Department of Computer and Information Sciences, University of Massachusetts, Amherst, 1987.

[8] Huff, K.E. and V.R. Lesser, "A Plan-based Intelligent Assistant That Supports the Process of Programming", Technical Report 87-09, Department of Computer and Information Sciences, University of Massachusetts, Amherst, 1987.

[9] Litman, D., "Plan Recognition and Discourse Analysis: An Integrated Approach for Understanding Dialogues", PhD. Dissertation (also TR 170), Department of Computer Science, University of Rochester, 1985.

[10] McDermott, D., "Planning and Acting", *Cognitive Science*, vol. 2, 1978, pp. 71-109.

[11] Parnas, D., "On the Criteria to be Used in Decomposing Systems into Modules", *Communications of the ACM*, vol. 15, no. 3, March 1972.

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

[13] Stefik, M., "Planning and Meta-planning", *Artificial Intelligence*, vol. 16, 1981, pp. 141-169.

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

[15] Wilensky, R., "Meta-Planning: Representation and Using Knowledge About Planning in Problem Solving and Natural Language Understanding", *Cognitive Science*, vol. 5, 1981, pp. 197-233.

[16] Wilkins, D.E., "Recovering From Execution Errors in SIPE", TN 346, SRI International, January, 1985.

This work was supported in part 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. This contract supports the Northeast Artificial Intelligence Consortium (NAIC).