HEWLETT PACKARD

Towards Self-Evolving Process-Driven Environments*

Sanjay Bhansali**, Pankaj K. Garg
Software Technology Laboratory
HPL-93-94
October, 1993

process-driven software engineering environments, process programming, algorithms, learning

Process-driven software development environments (PSDE's) provide support for the software (life-cycle) processes within which the environment is embedded and operational. This support includes tools and mechanisms for: modeling, analysis, automation, execution, optimization, and evolution of processes. In this paper, we concentrate on the mechanisms that a PSDE can provide for the evolution of software processes.

We envisage a PSDE with a repository of process programs that are applicable in different situations. The repository needs to be evolved with additional process programs when: (1) a new automatable process fragment is discovered, or (2) the programming tools or policies have changed. We discuss the evolution of this repository wherein the environment plays a helping role – it assists by generalizing the history sequence of user actions that led to the discovery of the new process fragment.

The contributions of this paper are: a formal model of a PSDE, and an explanation-based generalization algorithm that helps in a PSDE's self-evolution. For experimental purposes, the algorithm has been implemented for the UNIX programming environment, with specialized shell scripts as process programs. We present empirical evidence, based on this implementation, that suggests the practicality of our approach. In addition, we suggest that if shell scripts, i.e., process fragments, were developed using our approach then they will be more generally applicable and reusable.

*This is a revised and enhanced version of the paper, *Process Programming by Hindsight*, that appeared in the *Proceedings of the 14th International Conference on Software Engineering*, Melbourne, Australia, May 1992, and as HP Labs technical report HPL-92-31.
**Stanford University, Palo Alto California

Contents

1 Introduction **1**

1.1 Motivating Example 2

1.2 Background 4

1.3 Explanation Based Generalization 5

2 Process model **6**

3 The modified EBG algorithm **9**

4 The forced-check-in Example Revisited **12**

4.1 The example problem 12

5 Empirical Validation **17**

5.1 Quantitative Justification for synthesizing Shell Scripts 18

5.2 Applicability and Generality of approach 20

5.3 Qualitative Justification for synthesizing Shell Scripts 20

6 Summary and Future Work **21**

A UNIX Commands Modeled in the Environment **22**

B Representative Sample of Processes **29**

C References **33**

1 Introduction

A software development environment (SDE) provides computational support for the software development process by providing a collection of tools or operations, each of which performs a primitive action that can be used to achieve goals in some higher-level processes. Process-driven software development environments (PSDE's) seek to provide support for the automation, analysis, optimization, human understanding, communication, and execution of these higher-level processes. In this paper we concentrate on the need to provide automation support for the generation and evolution of software processes within PSDE's.

Towards the end of process automation, researchers have developed process programming languages (e.g., APPL/A [23] and HFSP [24]) and plan- or rule-based formalisms (e.g., GRAPPLE [12] and MARVEL [14]) that can be used to relate primitive actions (also called processing steps) to some processing goal in the form of a process program or plan. Since these languages are machine executable such process programs enable one to automate the execution of routine aspects of the software development process thus relieving some of the burden on the software developer.

Process programs embedded within process driven environments need to be evolved for two main reasons. First, it is difficult to *a priori* specify all process programs [15]. The difficulty arises from several factors including the non-availability of comprehensive application domain models, the skills of the various people involved, the degree to which the artifacts and resources of a software development effort have been institutionalized, and so on.

Second, process programs need to be evolved to keep up with changes in the environment. In other words, even when successful process programs have been developed, the advent of new tools and technology, and changing software practices and policies would require that these process programs be continually revised and updated.

In this paper we present an approach that provides automated support in the *evolution* of process programs in a PSDE. The approach is based on the intuition that even though process programs are difficult to specify *a priori* they are much easier to describe in *hindsight* after a particular sequence of processing steps has been successfully applied in a novel task or situation [9].

We envision, as part of a process-driven environment, a *process repository* (see Figure 1) that contains reusable *process fragments* annotated with conditions of their applicability and the goals achieved by the process fragment. The process fragments are composed from a set of atomic operations called *primitive actions*. These process fragments and the primitive actions are made available to a user by the process-driven environment. It is up to the user to make use of these in order to perform a more abstract, higher-level process. The process-driven environment automatically keeps track of the actions invoked by the user in a *work session* in the form of a *process history*. Given such an environment, most stages in a process could be characterized as belonging to one of two cases: Case 1, in which one or more pre-defined process fragment is applicable to the given situation, and case 2 in which a novel situation is encountered and none of the pre-defined process fragments is applicable. In case 1, the user can simply apply a relevant process fragment. In case 2, the user may continue to work by using the primitive actions available in the environment. Eventually, the user may realize that the situation and the subsequent sequence of actions taken is likely

to be repeated in the future. The user can use this *hindsight* to formulate the specific goal that was achieved by his actions. From this goal and the process history, a *process fragment generator* can help the user in creating a *general* process fragment which can be used in a class of situations. This process fragment would then be added to the reusable process repository and made available for future use.

1.1 Motivating Example

We introduce a simple motivating example to illustrate our approach; the example will also be used later to explain our technique for generating a process program.

Suppose there are two users U_1 and U_2 working off a shared pool of files, P_s. Suppose that U_1 has a working pool of files, P_1, and U_2 has a working pool of files, P_2. The normal mode of operation is that either U_1 or U_2 locks a file F_1 in P_s. If U_1 has locked F_1, then U_1 is allowed to make modifications to his copy of F_1 while U_2 has a read-only copy of F_1. Ideally, when U_1 has finished making his modifications, he creates a new version F'_1 of F_1 in P_s. At this point, U_2 has two options: she can either continue using F_1, or check out F'_1 from P_s.

One of the ways in which this normal process can break down is when both U_1 and U_2 have checked out a file F_1 and neither of them have locked the file. Assume that both U_1 and U_2 are allowed to make modifications[1] to their version of F_1. However, at some point, changes made in the two versions have to be consolidated. There are two alternative approaches to achieve this: (1) Use the changed version of either U_1 or U_2, or (2) Try to merge the changes made by both U_1 and U_2.

The steps carried out in Alternative 1 are:

1. Login into U_1's machine.
2. Go to pool P_1.
3. Move F_1 to F_1.cp
4. Lock F_1 in P_s and obtain the shared version of F_1.
5. Move F_1.cp to F_1.
6. Check in F_1 to P_s.
7. Login into U_2's machine.
8. Goto P_2.
9. Check out F_1.

This is an example of a process which might be repeated quite often in situations where U_1 and U_2 have forgotten to lock F_1 and have inadvertently made modifications to F_1. Thus, it would be useful to generalize the activity history into a process fragment, e.g., a UNIX shell script shown in figure 2.

This process fragment, call it `forced-check-in`, generalizes the history as follows: For each element in a given set of files, it determines if the shared version of that file is different from the user's version. If it is, then it checks if that file is locked by any user or not. If the file is not locked, then the script

[1] This is possible because typically version control systems cannot exercise control over the users' copy of the files which are maintained by the host's file system.

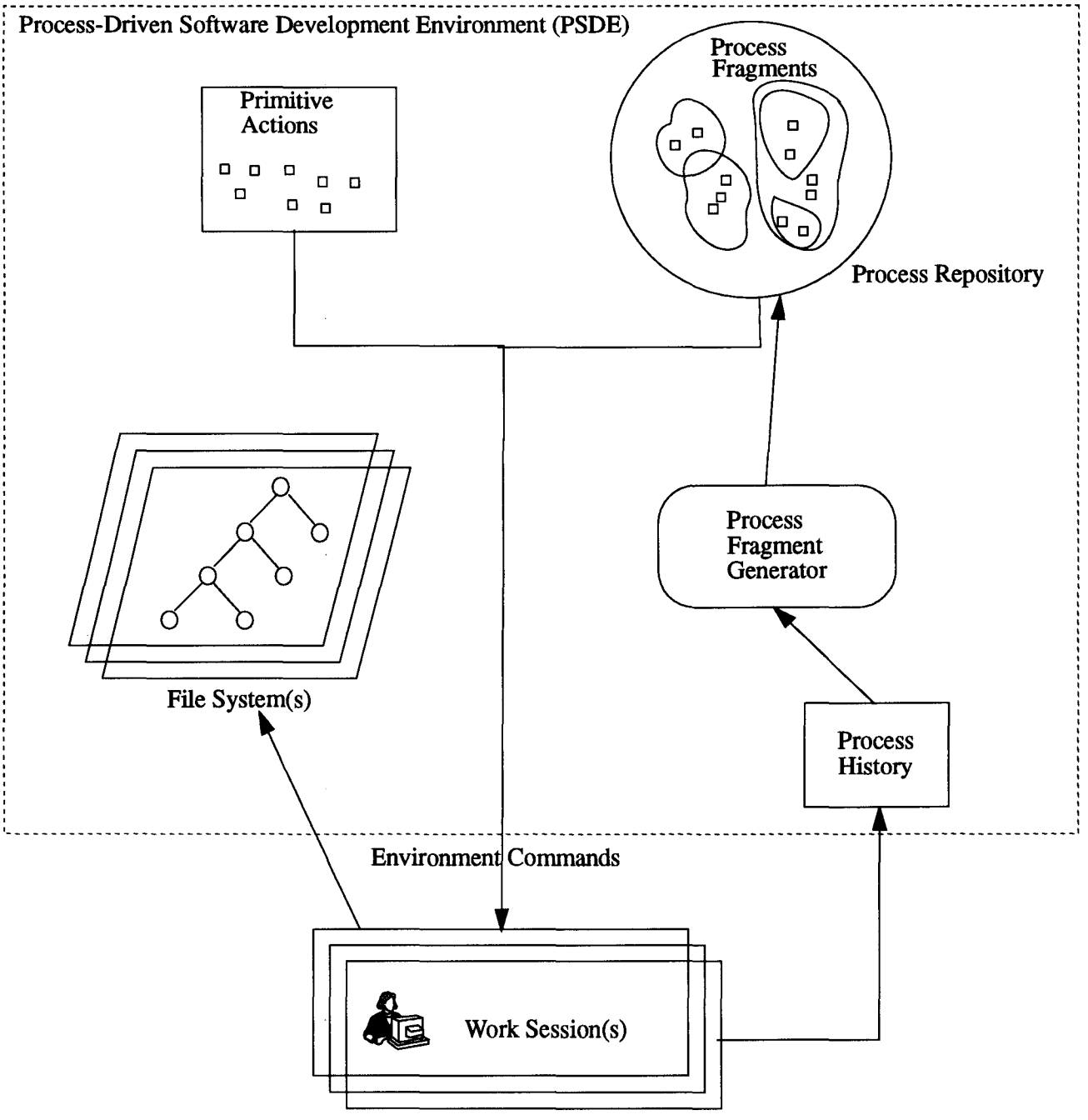

Figure 1: A Self-evolving Process-driven Software Development Environment

```
foreach file ($argv)
 set lock = `rlog -L $file`
 if ({ rcsdiff $file >& /dev/null }) then
   echo $file is same
 else
   echo $file is different
   if ( $lock == $file ) then
     echo $file is locked
   else
     mv $file $file.cp                                          10
     co -l $file
     mv $file.cp $file
     ci -u -m " " $file
     echo $file checked-in.
   endif
 endif
end
```

Figure 2: Example process fragment: `forced-check-in`

renames the user's version of the file. It then checks out the shared version of the file and locks the shared copy. Finally, it moves the changed copy which it had saved in a temporary location, to be used as the latest version of the file.

Our goal is to provide a mechanism that can provide automated support in creating such a process fragment by generalizing the process history.

1.2 Background

The idea of synthesizing a program from a trace of computations is quite old and dates back to the Autoprogrammer system developed by Biermann and Krishnaswamy [2]. The Autoprogrammer system was based on the idea of dividing the responsibility of program construction by assigning to a human the task of furnishing a trace of an algorithm on example problems and the system the task of generating code that is consistent with the examples.

In the past decade, the idea of reusing the trace of a problem solving activity has been successfully applied by researchers in program transformation, design, and plan-based problem solving in a variety of domains (e.g., [1, 3, 11, 13, 18, 19, 22, 26]). Instead of generalizing a problem solving history into a program, most of these systems (with the exception of [13]) simply record the history of a problem-solving activity. This history is then replayed to increase the efficiency of the problem solving process in a new, analogous context. Some of the earlier replay systems (e.g., [26, 18]) do not explicitly represent the goals of each step of the process; in such systems it is difficult to generalize a process script which can be automatically replayed in other situations and typically

such systems are used for design iteration. More recent systems (e.g., [1, 13, 3]) explicitly represent the goals of the problem-solving steps, and can obtain generalizations of a particular design history which can be used to solve several analogous problems.

In the process programming domain, the idea of viewing process programming as a mapping between explicitly represented goals and a sequence of environment actions was first proposed by Huff and Lesser [12]. Their system, GRAPPLE, uses the planning paradigm for *planning* (constructing a sequence of actions to achieve a given goal) as well as *plan recognition* (inferring a plan and its goal from a sequence of actions). The domain knowledge in GRAPPLE consists of a library of pre-defined higher level planning operators (e.g., *build* a system) which can be used to map the goal of building a system to a sequence of primitive operations (e.g., *unit-check-in, check-out*). A similar plan-based program synthesis approach is described by Bhansali and Harandi to synthesize UNIX shell scripts from user defined goals [1, 10].

However, for the same reasons that it is difficult to *a priori* specify algorithms for process programs, it is difficult to build a plan library with useful, pre-defined process plans. The approach we propose seeks to *learn* these plans by observing and analyzing a process history against a user-defined goal.

1.3 Explanation Based Generalization

Our approach is based on a well known learning technique called explanation-based generalization (EBG). Explanation based generalization is a technique to formulate a general concept on the basis of a specific training example. Our approach was inspired by the application of EBG to the problem of learning from untutored observation (e.g., [21, 27]). Such a learning situation enables a programmer to work unencumbered without having to articulate the rationale for each step of a process; the system unobtrusively observes the programmer's actions and learns by creating an explanation of how those actions satisfy a given goal. Of the several systems that learn from untutored observation, the one most similar to our work is the ARMS system developed by Segre [21] which learns sequences of manipulator motions of a robot arm. The algorithm described in this paper can be considered an adaptation of the algorithm used in ARMS, extended to handle a finer notion of operator (with conditional effects and different kinds of preconditions).

We first introduce some terminology from the planning and explanation-based learning literature. *Objects* are entities which may have certain properties and relations to other objects. A specification of all the objects and their properties and relations is termed a *world state.* There is a finite set of *operators* that map one world state into another by creating or deleting objects or modifying their properties or relations. An operator is specified by specifying the conditions under which it can be applied and its effects. A *problem* is specified by specifying a partial world state, called the *goal state* and another partial state called the *initial state.* A *plan* is a partially ordered set of operators, linked causally, and a *solution* to a problem is a plan, that transforms the initial problem state into a goal state.

The input to the EBG algorithm is the following:

- *Goal:* A specification of the goal state.

- *Initial State:* A specification of the initial state of objects and their properties.
- *Domain theory:* A set of inference rules and problem-solving operators that can be used to form a plan for achieving the goal state.
- *(Optional) Observed operator/state sequence:* A sequence of operator applications/ states which achieves an instance of the goal.

Given these four inputs, the task of the EBG algorithm is to determine a plan and a generalization of the plan that achieves the goal in a general way. The EBG algorithm consists of two stages:

Explain: If there is no observed operator input, construct a plan which achieves the goal state. Otherwise, build a causal explanation of how the operators in the input sequence achieve the goal.

Generalize: Determine a set of conditions under which the explanation holds. This is typically done by regressing the goal through the explanation structure using a modified version of the goal-regression algorithm described by Waldinger [25] and Nilsson [20]. The conjunction of the resulting expressions determine the conditions under which the generalized plan can be used to achieve the goal. See [7], [17], and [21] for further details.

These two steps can be summarized as follows: The first step creates an explanation structure that determines the relevant steps in a sequence of operations that are necessary to achieve the goal. The second step analyses this explanation to determine the constraints that are sufficient for this explanation to apply in general.

Since a process history may be defined as a sequence of low-level activities, one could use the EBG approach to create an explanation of how a subsequence of an arbitrary sequence of these activities achieves a specific goal, and then generalize this explanation to create a general process fragment which would be applicable in several different situations.

In the rest of this paper we will first define a formal model of a process driven environment. The model is not meant to be definitive – it is used to delineate the scope of our work and make the assumptions and limitations explicit. We then describe our algorithm as an extension of explanation-based generalization and illustrate the application of the algorithm on the `forced-check-in` example. Subsequently we discuss an empirical study that was performed to investigate the utility and generality of our approach. Finally, we conclude with a summary of the main contributions and suggestions for future work in this direction.

2 Process model

Although our approach is independent of any particular process-driven environment, we need a concrete environment in which to assess the utility of the approach. In order to make the examples accessible to a wider audience we have chosen the UNIX programming environment with commonly available version control systems such as RCS. The relevant objects in this domain are the generic objects in the UNIX environment including files, directories, strings, users, and the file-system.

The file system is a rooted directed acyclic graph of files. Each file in a file system can have a finite number of *versions.* The process programs are shell scripts.

A *condition* is a formula in first-order logic and can be in one of the following forms:

- $\alpha(\overline{x})$
- OR $(\alpha_1(\overline{x}), \alpha_2(\overline{x}))$
- $\forall \overline{x} \alpha(\overline{x})$

where α is a signed or unsigned predicate over the arguments $\overline{x}$. All free variables in a condition are existentially quantified.

A *state* is represented as a set of (conjunctive) conditions specifying the properties and relationships between objects.

A *primitive action* is an operation on state s_i to yield state s_{i+1}.

Definition 2.1 (Primitive Action) *A primitive action, A, is represented as a tuple* $< P_s, P_h, F, O, E, CE >$ *where:*

- P_s, P_h *(soft and hard preconditions, respectively) and* F *(filters) are conditions that must be true for A to be applicable in a given state,* S*. The difference between* P_s*,* P_h*, and* F *is as follows: if* P_s *(or* P_h*) is not true in* S*, then one may first create a sub-plan that transforms* S *into a state* S' *such that* P_s *(or* P_h*) is true in* S' *and then execute A, whereas if* F *is not true then A must be rejected, and an alternative action or plan must be found. If* P_s *cannot be satisfied, even then A can be applied (possibly with some undesirable side-effects*[2]*), while if* P_h *cannot be satisfied, A cannot be applied.*
- O *is an operation with 0 or more arguments.*
- E *(effect) is a set of conditions which are true in the state resulting from the application of A in* S*.*
- CE *(conditional effect) is a condition-effect pair* $< C_c, C_e >$*, such that if the condition* C_c *is true in state* S*, then the effect* C_e *will be true in the state resulting from the application of A.*

An example definition of such a primitive action is the following:

```
(defaction (copy ?f1 ?f2 ?p1 ?p2)
  :prehard (AND (= (contents ?p1 ?f1 1) ?k1)
                (file ?f1 ?p1)
```

[2]For example, the operation move(f_1, f_2) can be applied even though f_2 exists, but has the harmful side-effect of overwriting f_2.

```
              (readable ?f1 ?p1)
              (readable ?p1)
              (writable ?p2))
:presoft (not (file ?f2 ?p2))
:effect (AND (= (contents ?p2 ?f2 1) ?k1)
             (file ?f2 ?p2)
             (readable ?f2 ?p2))
:ceffect (((writable ?f1 ?p1)
           (writable ?f2 ?p2))))
```

The above action copies a file $?f1$ in pool $?p1$ to file $?f2$ in pool $?p2$. The hard preconditions for the action are that the file $?f1$ exists and is readable, the pool $?p1$ is readable, and the pool $?p2$ is writable. The soft precondition is that there does not already exist a file $?f2$ in pool $?p2$. The effect of the action is to create a file $?f2$ in $?p2$ which is readable and has the same contents as the file $f1$. In addition, if file $?f1$ is writable, then so is file $?f2$.

We make the simplifying assumption that a primitive action does not change the truth value of a predicate unless explicitly stated in the effects of the primitive action (this is the well-known *STRIPS assumption* [8]). In addition, we assume that a primitive action is instantaneous and there are no other agents in the world. This allows us to model time as a sequence of discrete points, where each point corresponds to a distinct state.

Definition 2.2 (Process History) *A* Process History *is a total order*

$$< \{O_1, O_2, \cdots, O_n\}, \leftarrow >$$

where each O_i is an operation with non-variable arguments and $\leftarrow$ is the "temporally-precedes" relation.

Definition 2.3 (Process Fragment) *A* Process Fragment, $\mathcal{P}$, *is a tuple $< P, \mathcal{O}, G >$, where:*

- *P (pre-condition) is a set of conditions which must be true in a state for $\mathcal{P}$ to be applicable.*
- *$\mathcal{O}$ is a partial order:*

  $$< \{O_1, O_2, \cdots, O_n\}, \leftarrow >$$

  where each O_i is an operation (with variable or non-variable arguments).
- *G (Goal) is a set of conditions that are true in the state resulting from the application of any total ordering of the operations comprising $\mathcal{O}$.*

Definition 2.4 (Domain Theory) *A domain theory, $\mathcal{D}$ is specified as a tuple $< N, P, F, \mathcal{A} >$, where:*

- *N is a set of* Objects.

- *P is a set of predicates on N.*
- *F is a set of functions which take arguments from N.*
- *A is a set of primitive actions.*

The generalization problem that we are interested in, can now be summarized as follows:

Given:

- A domain theory.
- A goal-state (or post-condition) and an initial-state;
- An observed process history that is known to achieve the post-condition;

Determine:

- A process fragment that achieves the post-condition in a general way.

3 The modified EBG algorithm

The algorithm we use for generating a process fragment from a process history is an extension of the general EBG algorithm mentioned earlier, and consists of two steps:

- *Explanation construction:* Construct an explanation to show how the process history achieves the given post-conditions.
- *Generalization:* Generalize the explanation to determine a process fragment and a set of conditions that are sufficient for the process fragment to achieve a generalized post-condition.

We introduce some notation necessary to describe the explanation construction process.

Definition 3.1 (Satisfies) Satisfies(α, β) *is defined recursively as follows:*

- *satisfies*(α, α)
- *satisfies*$(\forall \overline{x}\ \alpha(\overline{x}),\ [\alpha(\overline{x})]_{\overline{x}=\overline{C}})$
- *satisfies*$(\alpha,\ OR(\beta_1, \beta_2))$ *if satisfies*(α, β_1) *or satisfies*(α, β_2)

where $\overline{x}$ denotes a set of variables and $[\alpha(\overline{x})]_{\overline{x}=\overline{C}}$ denotes a formula in which all occurrences of variables $\overline{x}$ have been replaced by constants $\overline{C}$. Analogously, we define *denies*(α, β):

Definition 3.2 (Denies) Denies(α, β) *is defined recursively as follows:*

- *denies*$(\gamma, \neg\gamma)$
- *denies*$(\neg\gamma, \gamma)$
- *denies*$(\forall\overline{x}\ \alpha(\overline{x}),\ [\neg\alpha(\overline{x})]_{\overline{x}=\overline{C}})$
- *denies*$(\forall\overline{x}\ \neg\alpha(\overline{x}),\ [\alpha(\overline{x})]_{\overline{x}=\overline{C}})$
- *denies*$([\neg\alpha(\overline{x})]_{\overline{x}=\overline{C}},\ \forall\overline{x}\ \alpha(\overline{x}))$
- *denies*$([\alpha(\overline{x})]_{\overline{x}=\overline{C}},\ \forall\overline{x}\ \neg\alpha(\overline{x}))$
- *denies*$(\alpha,\ OR(\beta_1,\ \beta_2))$ *if denies*$(\alpha,\ \beta_1)$ *and denies*$(\alpha,\ \beta_2)$

Here γ represents an arbitrary condition. We use σ_i to denote the substitution of all free variables in the effects and preconditions of the primitive action A_i. Similarly, we use E_i to denote the effects, CE_i to denote the conditional effects and PC_i to denote the filter and hard preconditions of a primitive action A_i.

We will describe the explanation construction algorithm using the notion of supported effects and supported preconditions of a primitive action. Informally, if an effect is supported then it can be used to infer predicates that form the preconditions of subsequent primitive actions in a process history. A precondition is supported if it can be inferred from the effects of a set of preceding primitive actions. We assume that each process history begins with a primitive action *start* which has no preconditions and whose effect is the INITIAL-STATE and ends with a primitive action *end* which has no effect, and whose precondition is the GOAL-STATE.

Definition 3.3 (Supported Effect) *E_i is supported if all PC_i's are supported.*

Definition 3.4 (Supported Conditional effect) *Let $< C_c,\ C_e > \in CE_i$. C_e is supported if all PC_i's are supported and C_c is supported.*

The definition of supported preconditions is based on the *modal truth criterion* in [4].

Definition 3.5 (Supported Precondition) *Let $C \in PC_i$. C is supported if:*
(1) there exists $E \in E_j$, $A_j \leftarrow A_i$, E_j is supported, $E\sigma_j$ satisfies $C\sigma_i$,
OR
there exists $< C_c,\ C_e > \in CE_j$, $A_j \leftarrow A_i$, C_e is supported, $C_e\sigma_j$ satisfies $C\sigma_i$, and
(2) there does not exist E', E_k, such that $A_j \leftarrow A_k \leftarrow A_i$, $E' \in E_k$, E_k is supported, and $E'\sigma_k$ denies $C\sigma_i$, and
(3) there does not exist $< C_c,\ C_e >$, CE_k such that $< C_c,\ C_e > \in CE_k$, $A_j \leftarrow A_k \leftarrow A_i$, C_e is supported, and $C_e\sigma_k$ denies $C\sigma_i$.

A *clobbered precondition*, which refers to a precondition known to be false in a state, is defined analogously as above (substituting *clobbered* for *supported*, *denies* for *satisfies*, and *satisfies* for *denies*). A precondition that is neither supported or clobbered is said to be unsupported.

Definition 3.6 (Conditionally supported) *Let $E \in E_i$. E is conditionally supported if none of the PC_i's are clobbered and there is atleast one $C \in PC_i$ that is unsupported. C is called a supporting-condition of E.*

Let $< C_c, C_e > \in CE_i$. C_e is conditionally supported if none of the PC_i's or C_c is clobbered, and at least one of $C \in PC_i$ or C_c is unsupported.

The explanation construction algorithm can now be simply described as determining a set of substitutions of the free variables in the primitive actions belonging to the process history, such that each condition in the goal-state is supported. The network of all the supporting links used in determining the set of supports for the goal conditions constitutes an explanation. If there is no explanation in which none of the conditions in the goal-state is clobbered, then the explanation phase fails. (Thus, an explanation is considered valid even if some of the goal conditions are not supported, as long as none of them is clobbered.) Otherwise, the explanation with the maximum number of supported goals is selected and is generalized in the second stage of the algorithm.

The generalization algorithm begins with generalizing the postconditions of the problem (replacing all the constants by variables) as far as possible while ensuring the correctness of the supporting links. This algorithm is similar to the classical EBG generalization and consists of matching an expression with the effect of an operator to yield a set of substitutions; the substitution which preserves the validity of the explanation is then applied to the preconditions of the rule to yield a new set of regressed expressions. Preconditions that are not supported are simply added to the regressed expression (with the substitutions applied to them). The set of regressed expressions left at the end become the preconditions for the process fragment.

Next, starting with the generalized preconditions, the algorithm rederives the explanation to obtain the generalized goals. See [7] for an explanation of why this step is necessary. Intuitively, it is needed to ensure that we do not overgeneralize the postconditions of the the problem.

Finally, the EBG algorithm derives the ordering constraints between primitive actions. We will say that a primitive action A_i supports another primitive action A_j using E_i if E_i is an effect/conditional-effect of A_i that supports a precondition of A_j. The ordering constraints are determined as follows:

(1) If A_i supports A_j then $A_i \leftarrow A_j$.

(2) If A_i supports A_j using E_i, and there exists $E' \in E_k$ such that $E'\sigma$ denies E_i for some substitution σ, then either $A_k \leftarrow A_i$ or $A_j \leftarrow A_k$. (In planning terminology, the interval between the states just after A_i and just before A_j is called a *protection interval* for E_i. E' is a conflicting action and the conflict is resolved by placing E' either before or after the protection interval.)

Once these constraints are determined, it can be used to determine a generalized process fragment in the form of a shell script.

4 The forced-check-in Example Revisited

We now illustrate the application of the modified EBG algorithm to generate a process fragment for the `forced-check-in` example given in section 1.1. We first describe the relevant domain theory, $\mathcal{D}$.

N consists of a set of symbols denoting users, components of the File System, the set of integers, and the set of arbitrary text strings.

P contains the following predicates:

file(f,p) which is true iff f is a leaf node and p is an internal node and there is an edge from p to f in the File System.

locked(u, f, p_s, p, n) where u is a user, f is a leaf node, p_s, and p are internal nodes, n is an integer, and the n^{th} version of f in p_s can be modified only by user u.

F contains the following functions:

contents(p, f, m) whose arguments are an internal node, p, of the File System, a leaf node, f, of the file system and an integer, m, and whose output is a text string. If f does not exist under p, *contents*(p, f, m) returns a special symbol nil.

last_version(p, f) which returns the last version of the leaf node f in p.

We assume the availability of an '=' predicate for arbitrary text strings. We also assume that all versions of a file can be put into one total ordering according to their creation time.

The set, A, of primitive actions used in our example consist of *move, lock, check-in, and check-out* (defined in Appendix 1).

4.1 The example problem

The initial state and the goal state (or postcondition) of our example problem is:

GOAL-STATE:

1. *contents*$(P_1, F_1, 1)$ = *contents*$(P_s, F_1, M + 1)$
2. $M + 1$ = *last-version*(P_s, F_1)
3. *contents*$(P_1, F_1, 1) = C$

INITIAL-STATE:

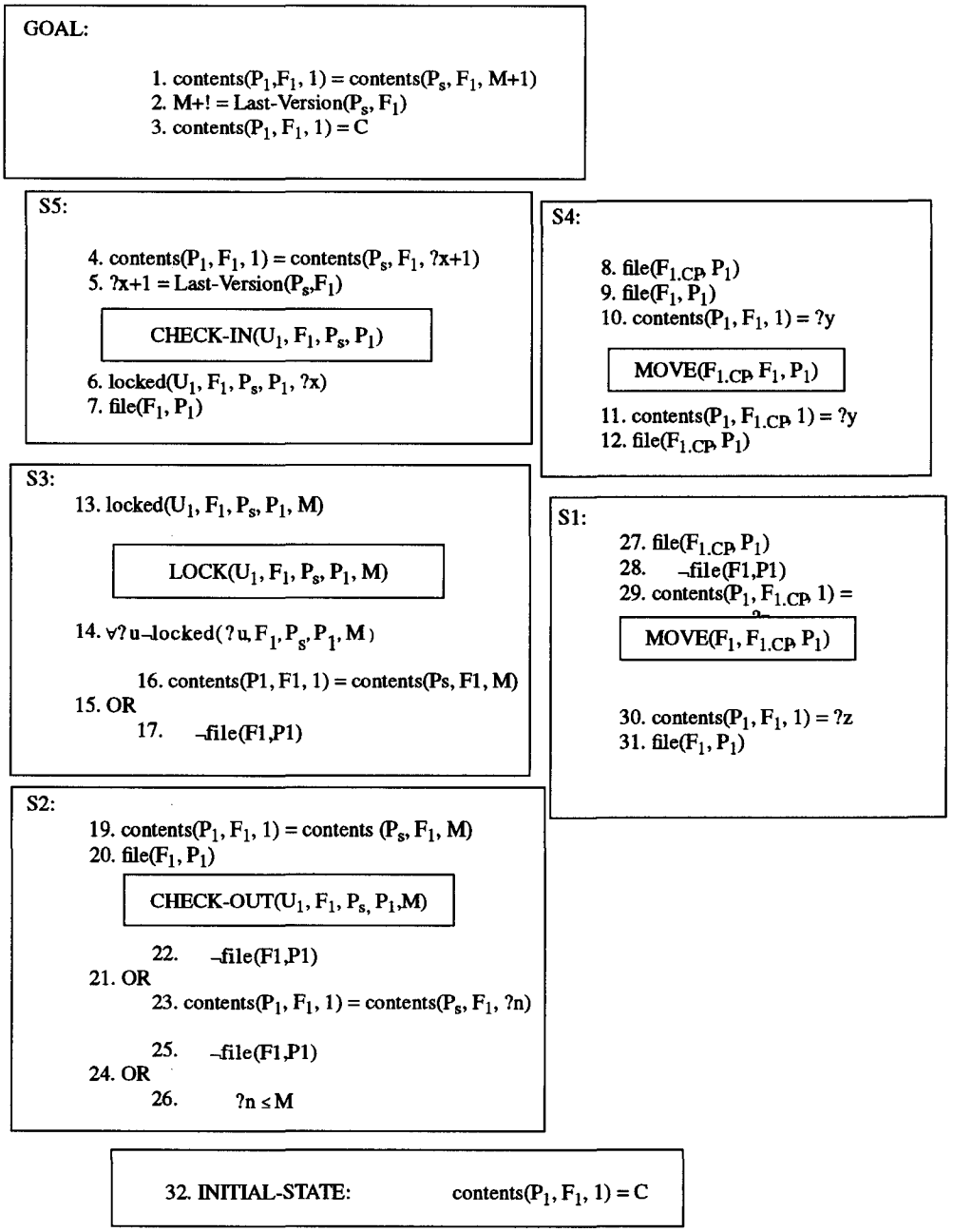

Figure 3: The process history of the training example. S1-S5 are the states resulting after each primitive action in the history.

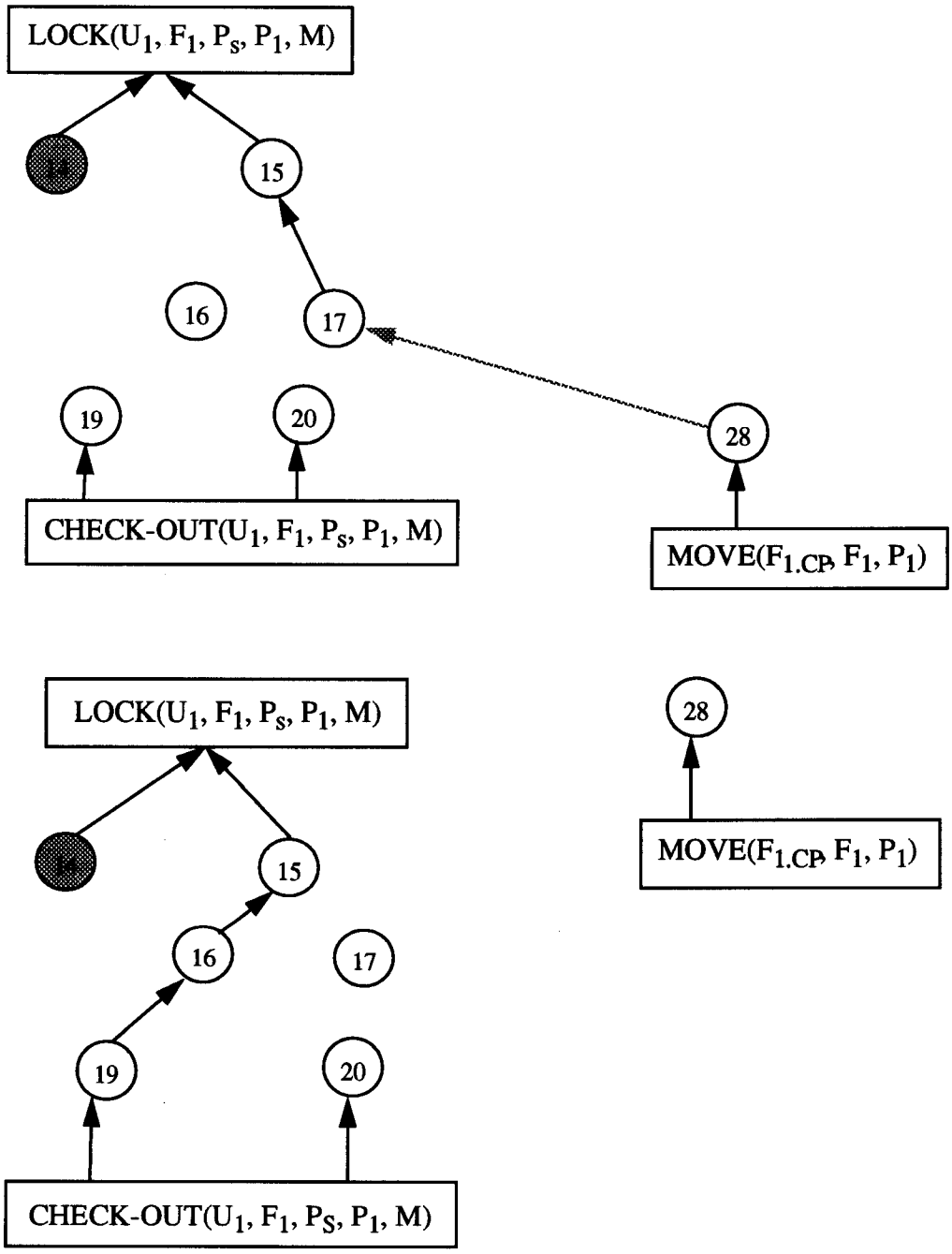

Figure 4: An example showing precondition-clobbering. Precondition 15 which represents a disjunction can be satisfied either by 16 or 17. 17 has a support from effect 28. However, this gets clobbered by effect 20 of the CHECK-OUT action, which lies in the interval from MOVE($F_1.CP, F_1, P_1$) to LOCK(U_1, F_1, P_S, P_1, M). The correct explanation is through 16 which is supported by effect 19 of CHECK-OUT.

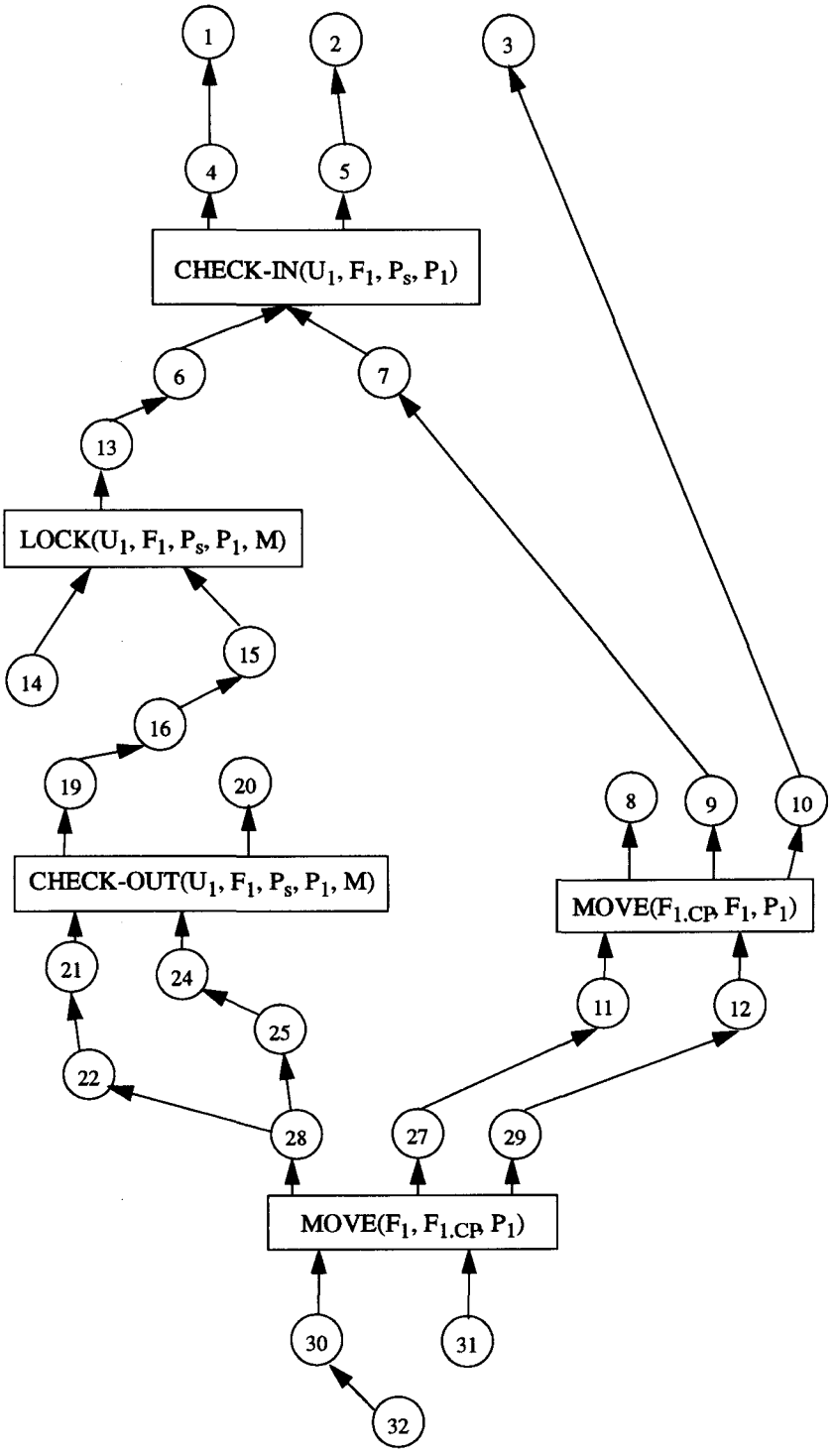

Figure 5: The complete explanation structure for the example. The numbers in the figure refer to conditions in figure 2

1. $contents(P_1, F_1, 1) = C$

Figure 3 shows the process history containing the 5 primitive actions and the initial and goal states. Since each primitive action in the initial process history is a ground action (i.e., having non-variable) arguments, the substitution σ_i is nil for most primitive actions. The substitution for the remaining variables is obtained by determining the most specific substitution needed to satisfy the preconditions of a supported primitive action. For example, the substitution $\{ ?x = M\}$ is used to form a support from effect 5 to the goal condition $M+1 = last\text{-}version(P_s, F_1)$.

The example also illustrates an instance of precondition clobbering shown in figure 4. The complete explanation structure generated after the explanation stage is shown in figure 5. There are two preconditions that are unsupported (condition 14 and 31). Whenever the system is unable to generate a complete proof, it queries the user for the truth value of conditions that would enable it to complete the proof. Thus, in the above explanation, the system generates the following two queries during the proof:

- Is the following condition true in the initial state:
  $\forall$?u $\neg$(locked($?u, F_1, P_S, P_1, M$)?
- Is the following condition true in the initial state:
  (file F_1,P_1)?

If the user answers *yes* to the above queries, the system marks them as assumptions and continues with the proof. Note that because of this interactive nature of the proof generation, it is not necessary for a user to give a complete specification of the initial state; the missing conditions are automatically identified and verified during the proof. Thus, in the above program even when no initial conditions are specified, the system can generate the initial state that would explain the goal (see Appendix 2).

Next, the explanation is generalized using goal regression. The generalized preconditions left at the end of this step are (all lowercase arguments denote variables):

- $\forall$ u $\neg$(locked(u, f_1, p_S, p_1, x)
- file(f_1, p_1)
- contents($p_1, f_1, 1$) = c

The free variables in the above preconditions form the parameters for the generalized process fragment.

Finally, the system generates the ordering constraints between the primitive arguments. The final process fragment $< P, \mathcal{O}, \mathcal{G} >$ that is generated for this example is:

- $P = \forall u \neg$ (locked(u, f_1, p_S, p_1, x) $\wedge$ file(f_1, p_1) $\wedge$ contents($p_1, f_1, 1$) = c

- $\mathcal{O}$ = <$\{A_1, A_2, A_3, A_4, A_5\}$,$\leftarrow$> where:

  A_1 = MOVE(f_1, f_2, p_1, p_1)
  A_2 = CHECK-OUT(u_1, f_1, p_S, p_1, x)
  A_3 = LOCK(u_1, f_1, p_S, p_1, x)
  A_4 = MOVE(f_2, f_1, p_1, p_1) }
  A_5 = CHECK-IN(u_1, f_1, p_S, p_1)
  $\leftarrow$ = $\{(A_1, A_2)(A_2, A_3)(A_2, A_4)(A_3, A_5)(A_4, A_5)\}$

- G = contents(p_1, f_1,1) = contents(p_S, f_1,x+1) $\wedge$
  x+1 = last-version(p_S, f_1) $\wedge$
  contents(p_1, f_1,1) = c

The system then prompts the user for an annotation for the processs program in terms of the free variables in the goal and initial state:

System: Name of the process program:

User: forced-check-in

System: Give an annotation for the process program in terms of the variables: f_1, p_1, f_2, p_s.

User: (Given a file, f_1 in a pool p_1, make the last version of f_1 in p_s equal to f_1, without changing the contents of f_1 in p_1. Use f_2 as a temporary file name.)

When this generalized process program is to be reused, the system displays the annotation associated with the process program and asks the user to provide values for instantiating the free variables in the annotation. It then instantiates the process program with the given values and executes the process program.

5 Empirical Validation

We have implemented the above algorithm fully and have used it to successfully synthesize several shell scripts which gives us confidence in the robustness of the implementation. The variety of shell scripts that we could synthesize also indicated the potential for the practicality of our approach. Our next goal was to obtain evidence to empirically assess the practicality of our approach. Towards this end we have conducted some experiments using the UNIX computing environment. Even though the UNIX environment is limited in its support for process programming, it provides a Shell language (e.g., csh), that can be used to write process programs (shell scripts or simply scripts).

The ideal technique for evaluating our approach would be: (1) build a system that records users' actions in the UNIX environment and guides them in building process programs, and (2) monitor the system's effectiveness in evolving a set of process programs through actual usage. However, such an experiment is expensive and could be be in vain if a negative result is obtained. An alternative

approach is to evaluate existing Shell scripts that people have written and retrospectively evaluate the benefits of our approach. Such an experiment can be used to justify the expense involved in conducting the former experiment.

We have conducted the second experiment with three questions in mind:

- How useful are the process fragments/shell scripts that could be generated by our system in the context of an overall process model?
- How many of such process programs could have been synthesized from a passively acquired history of user actions?
- What are the benefits of using our approach versus the current approach, i.e. the user foresees the repeated need for a particular process at some point and manually writes and debugs a script for it?

The first question helps us in obtaining a quantitative justification for using our approach - if there are very few useful shell scripts that can be generated within our framework then there would be little point in continuing with this approach. The second question helps us in determining the technical feasibility of generating process fragments using our approach. As a side-effect, it also points out the limitations of the approach and directions for further work in order to overcome these technical limitations. The third experiment helps us in obtaining qualitative justification for (or against) using the approach. For example, if the shell scripts generated using our generator are qualitatively superior to those obtained using current techniques, then one might still be justified in pursuing this approach. Note that we expect the answers for the first two questions to be more favorable for a software development environment that has specifically been designed to be process-driven.

5.1 Quantitative Justification for synthesizing Shell Scripts

Figure 6 shows the space of all process fragments comprising the software engineering activity. From a process-oriented perspective, the processes can be grouped into two broad categories - those that can be represented as a machine-executable program in some computational formalism (e.g., plans, rules, or programs in some process-oriented language) and those that cannot (e.g., processes that consist primarily of human interaction). The boundary between these two kinds of processes is fluid and we expect that as process-driven software environments mature, the fraction of machine-executable software engineering processes will increase. The goal of our first experiment was to estimate what fraction of the executable process fragments could be synthesized using our approach.

For the experiment we analyzed 199 Shell scripts collected from two experienced UNIX users. We identified three interesting categories into which the various shell scripts could be placed from a process-modeling perspective:

1. *Process programs:* These are scripts that can be specified using the primitives available in the *domain theory* (section 2).

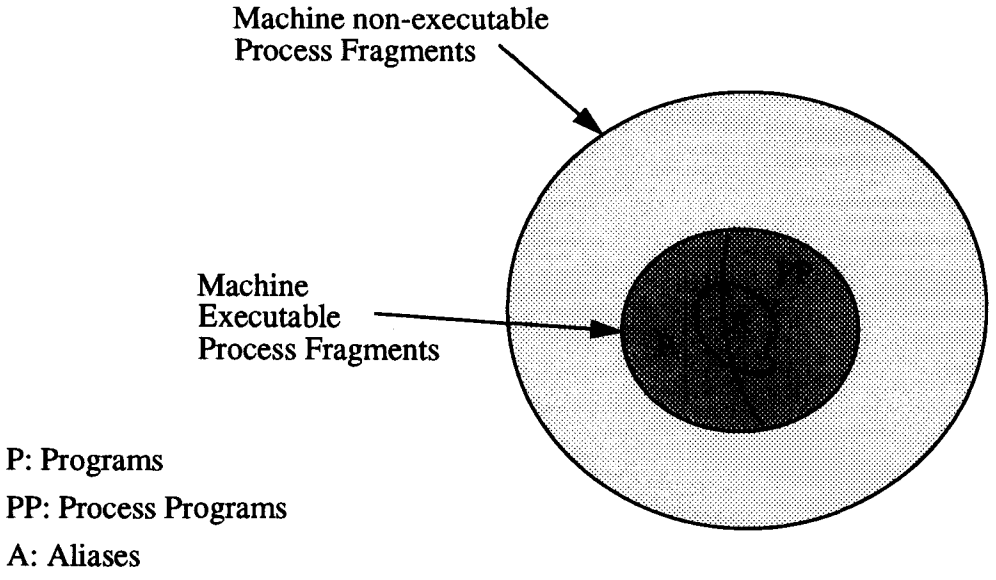

Figure 6: Space of Software Process Fragments

2. *Programs:* These are scripts that cannot be specified using the primitives available in the domain theory.

3. *Aliases:* These are simple customization of tool invocations and could be either process programs or programs. The distinguishing feature of aliases is that they consist of just one or two commands that get called with the same set of arguments.

Based on these definitions, the classification of a script as a process program or a program depends on the knowledge encoded in a domain theory. Increasing the amount of knowledge leads to a corresponding increase in the number of scripts that can be classified as process programs. The following example should make this point clear: Suppose there is a file, call it `project-members-address-file`, in which the names, e-mail addresses, and phone numbers of members of a software project are maintained in a specific format. One may write a shell script that, given a name, extracts the telephone number(s) of all persons with the given name from the `project-members-address-file`. If the `project-members-address-file` is modeled as part of the domain theory with a description of how the file is formatted in terms of records and the fields for each record, together with operators that can extract particular fields from the record (e.g., the `awk` operator in UNIX) then the above script would be a process program. Otherwise, it would be considered a program.

Clearly, scripts that are programs cannot be generalized by our approach and so we discounted

them as beyond the scope of our work. The other two categories - process programs and aliases - were considered potential candidates to benefit from the approach. *We found that about 80% of all scripts belonged to categories 1 or 3.* This gives us a sound reason to continue with the experiment based on shell script usage.

A second, perhaps more important observation, was that shell scripts that were classified as process programs were those that are potentially reusable in many different contexts and by many different users. On the other hand, shell scripts that were classified as programs were those that had been customized to perform a very specific task. Such scripts are usually of interest only to a small set of individuals (often just the writer of the script) and/or are applicable within a very narrow context. This addresses one of the potential limitations of the approach - the difficulty of identifying and representing the relevant domain theory for the EBG algorithm. Our observation suggests that a large number of the most reusable scripts can be generated by a relatively small domain theory.

5.2 Applicability and Generality of approach

For the second part of the experiment, we analyzed the scripts belonging to categories 1 and 3. We found that 93% of the analyzed scripts could have been synthesized based on a history of user actions. Appendix 2 shows a representative sample of some of these process programs.

The scripts that could not be generalized were of three kinds: (1) scripts whose goals could not be expressed within our language, i.e., as a conjunct of simple conditions; (2) scripts that involved the *if-then-else* construct in the script; and (3) scripts that contained loops. Scripts that contained *if-then-else* require the user to give two or more examples of process histories that differ in the condition of the *if-then-else* construct. Since one of our goals was to unobtrusively acquire process fragments by trying to explain and generalize what a user does in the normal course of his work, rather than ask him to provide specific examples, we do not consider such scripts in our approach. Moreover, such scripts can sometimes be represented as two different scripts having different preconditions for their applicability – therefore this is not a serious limitation.

On the other hand, scripts containing loops are quite useful. The generalization mechanism for such scripts is quite different from our approach. Instead of *deductive* reasoning (employed in explanation-based generalization) the synthesis of such scripts requires *induction.* This kind of generalization is performed by the EAGER [6] system. We plan to extend our algorithm by incorporating loop generalizations.

5.3 Qualitative Justification for synthesizing Shell Scripts

The main benefit that we perceive in using an automated approach like ours is that it reduces errors and the consequent effort in debugging and testing shell scripts that are written manually.[3] Since our experiment was based on already synthesized (and presumed correct) shell scripts there was no way of validating this claim directly. However, we could indirectly obtain evidence to support this claim by comparing each shell script with a corresponding script that was synthesized by our system.

[3]The need for testing is not entirely eliminated since the original specification provided by a user may not be correct and hence the user may still need to verify that the script's behavior agrees with the desired requirements.

We did the comparison based on two features: *pre-condition determination* and *reusability.*

One of the features of our appraoch is that it automatically enables one to determine pre-conditions for the applicability of the scripts. This ensures that the script is not inadvertently used in a wrong context. We found that about 27% of the sample scripts (belonging to categories 1 and 3) did not have appropriate pre-conditions defined. This implies that users of such scripts have to be quite careful – they may have undesirable side-effects, or they may not be utilized in situations were a small action to fulfill a pre-condition could have made the script applicable. These scripts could have been improved (from a pre-conditions viewpoint) had our approach been used to synthesize them.

Scripts that are written by people sometimes have limited reusability because they are not adequately parameterized. For example, names of particular files are hard-wired in the script. Our approach improves the reusability of scripts by providing the maximal possible generalization (of constants appearing in a process history). Of course, it is possible to over-generalize and it is debatable whether people actually require such generalizations. An argument against over-generalization is that it is cumbersome to have to instantiate each parameter before the script can be used. This is particularly true when the script is going to be used only within a very narrow context. However, the benefit of our approach is that the system automatically *identifies* the maximal generalization; it is then quite easy to customize the script for individual users (e.g., by providing default values for certain parameters or hard-wiring them). We found that many of the scripts we analyzed could have been made more general. However, we cannot provide any quantitative results without performing a cost-benefit analysis of such generalizations based on actual usage.

There are other routine automation benefits which we envisage from our approach – automatically pre-pending and post-pending history sequences with fixed templates, renaming files, cataloging scripts, etc. The real benefit of such aspects of our approach is user dependent. In some cases it might make the difference between a user using sripts versus not using them, in other cases it may not matter. A systematic cost-benefit analysis of such factors requires actual usage experiments which we leave for future work.

6 Summary and Future Work

In this work we have addressed the issue of evolution of process-driven software development environments. In our approach, the evolution is supported by not requiring an *a priori* specification of all possible process fragments applicable within the environment. Instead, the environment monitors users' activities and if a situation arises, where a sequence of activities may potentially be useful as a process fragment, the environment helps the user in generalizing the history sequence into a process fragment. The environment, therefore, helps in its own evolution. This approach is based on the intuition that process fragments are easier to describe in hindsight rather than by foresight. We have described an adaptation of the well-known explanation based learning algorithm to synthesize a process fragment from a process history. We have shown the generality and potential utility of this approach through empirical studies of shell scripts written for the UNIX programming environment.

The approach has certain limitations. Some of these require minor modifications whereas others

are open issues indicating directions for future work. In our work, we have focused on how useful process fragments can be acquired to build a process repository. An issue that we have not addressed is how to organise and retrieve the generalized process fragments in the process repository. One technique is to use higher-level process abstractions to organize the process fragments [16]. We believe that in a complete process-driven environment, higher level process model definitions will be available within the environment for such purposes. Moreoever, most of the process programs that are acquired using our approach will be specific to particular users and would reside in their private repositories. We believe that the use of meaningful, user-specified names and annotations would largely be adequate for such usage.

In order to make the system practical a user-friendly interface needs to be built. The minimal requirement for an interface would be a front-end that can translate a user's actions into the internal notation used in our process model. Additional capabilities of the interface would be to assist a user in formulating the goals of a process program formally. Rudimentary capabilities that can be currently implemented include providing a list of relevant predicates and functions to a user based on the most recent history of operations performed by the user. More advanced capabilities would include building a front end to translate a user's goal expressed in restricted, natural language into the logical notation understood by the system.

Our system may be viewed as a tool that can assist a user in verifying the correctness and in generalizing a sequence of processing steps. It is most likely to be useful for mechanizing frequently performed, general processes. The approach would not be very practical for acquiring process fragments that require detailed application specific knowledge, e.g., processes specific to requirements elicitation or design construction. Other techniques (based, for example, on an IBIS style design rationale representation [5]) might be more appropriate for such processes. We believe that eventually a variety of different tools, based on different techniques and representations, would be needed to provide automated support for the execution of processes spanning the entire software development lifecycle. Our approach provides one tool in such a repertoire.

A UNIX Commands Modeled in the Environment

The following are the (28) UNIX commands that we modeled in order to perform our empirical validation. Each command is preceded by a comment paraphrasing the effects of the command.

Move. *Move a file ?f1 in pool ?p1 to file ?f2 in pool ?p2.*

```
(defaction (move ?f1 ?f2 ?p1 ?p2)
  :prehard (AND (= (contents ?p1 ?f1 1) ?k1)
                (writable ?p2)
                (readable ?f1 ?p)
                (writable ?f1 ?p)
                (file ?f1 ?p1))
```

```
  :presoft (NOT (file ?f2 ?p2))
  :effect (AND (NOT (file ?f1 ?p1))
               (= (contents ?p ?f2 1) ?k1)
               (file ?f2 ?p2)
               (readable ?f2 ?p2)
               (writable ?f2 ?p2)))
```

Copy. *Copy a file ?f1 in pool ?p1 to ?f2 in pool ?p2.*

```
(defaction (copy ?f1 ?f2 ?p1 ?p2)
  :prehard (AND (= (contents ?p1 ?f1 1) ?k1)
                (file ?f1 ?p1)
                (readable ?f1 ?p1)
                (readable ?p1)
                (writable ?p2))
  :presoft (NOT (file ?f2 ?p2))
  :effect (AND (= (contents ?p2 ?f2 1) ?k1)
               (file ?f2 ?p2)
               (readable ?f2 ?p2))
  :ceffect (((writable ?f1 ?p1)
             (writable ?f2 ?p2))))
```

Lock. *Lock the ?nth version of a file ?f in pool ?p. ?v is the user doing the lock and ?ps is the shared pool.*

```
(defaction (lock ?v ?f ?ps ?p ?n)
  :prehard (OR (= (contents ?p ?f 1) (contents ?ps ?f ?n))
               (NOT (file ?f ?p)))
  :filter (FORALL (?u) (NOT (locked ?u ?f ?ps ?n)))
  :effect (locked ?v ?f ?ps ?p ?n)
)
```

Check-in. *Check in a file ?f from pool* $?p_{from}$ *to pool* $?p_{to}$ *by user ?u.*

```
(defaction (check-in ?u ?f ?p_from ?p_to)
  :prehard (AND (file ?f ?p_from)
                (locked ?u ?f ?p_to ?p_from ?m))
  :effect (AND (= (contents ?p_from ?f 1) (contents ?p_to ?f (+ ?m 1)))
               (= (+ ?m 1) (last-version ?p_to ?f)))
)
```

Check-out. *Check out the ?mth version of file ?f from pool ?p_{from} to pool ?p_{to} by user ?u.*

```
(defaction (check-out ?u ?f ?p_from ?p_to ?m)
  :prehard (AND (OR (NOT (file ?f ?p_to))
                    (= (contents ?p_to ?f 1) (contents ?p_from ?f ?n)))
                (OR (NOT (file ?f ?p_to))
                    (leq ?n ?m)))
  :effect (AND (file ?f ?p_to)
               (= (contents ?p_to ?f 1)
                  (contents ?p_from ?f ?m)))
)
```

Delete-file. *Delete a file ?f in pool ?p.*

```
(defaction (remove-file ?f ?p)
  :prehard (AND (writable ?f ?p) (writable ?p))
  :presoft (file ?f ?p)
  :effect (not (file ?f ?p)))
```

Catenate. *Catenate a file ?f in pool ?p to another file ?out Note that we treat files and streams interchangeably in our model. In particular, the standard output and standard input to/from the terminal are both treated as files. This also allows us to handle the pipe mechanism (|), the redirection operators (>, <) cleanly.*

```
(defaction (cat ?f ?p ?out)
  :prehard (AND (= (contents ?p ?f 1) ?k1)
                (file ?f ?p)
                (readable ?f ?p)
                (readable ?p)
                (writable ?out ?p))
  :effect (and (= (contents ?p ?out 1) ?k1)
               (file ?out ?p)
               (readable ?out ?p)))
```

Append. *Append file ?f out ?out in pool ?p. This models the >> operator in UNIX.*

```
(defaction (append ?f ?p ?out)
  :prehard (AND (= (contents ?p ?f 1) ?k1)
                (= (contents ?p ?out 1) ?k2)
                (file ?f ?p)
                (readable ?f ?p)
                (readable ?p)
                (writable ?out ?p))
  :effect (and (= (contents ?p ?out 1) (append ?k2 ?k1))
               (readable ?out ?p)))
```

Change-directory. *Change the current directory to ?p. Note, that we have not modeled the user explicitly and it is assumed to be a constant. A more detailed model would explicitly model the user.*

```
(defaction (change-directory ?p)
  :prehard (readable ?p)
  :effect (in-directory ?p))
```

Change-mode. *Make a file ?f in pool ?p writable. There are analogous commands to make the file readable and executable.*

```
(defaction (change-mode+w ?f ?p)
  :prehard (AND (file ?f ?p)
                (writable ?p))
  :effect (writable ?f ?p))
```

Change-mode. *Make a file ?f in pool ?p non-writable. There are analogous commands to make the file non-readable and non-executable.*

```
(defaction (change-mode-w ?f ?p)
  :prehard (AND (file ?f ?p)
                (writable ?p))
  :effect (not (writable ?f ?p)))
```

Echo. *Copy ?arg into file/stream ?out in pool ?p.*

```
(defaction (echo ?arg ?out ?p)
  :prehard (writable ?out ?p)
  :effect (= (contents ?p ?out 1) ?arg))
```

Expand. *Make the contents of file ?out equal to the contents of file ?f of pool ?p except that the Tabs are replaced by spaces.*

```
(defaction (expand ?f ?p ?out)
  :prehard (AND (file ?f ?p)
                (readable ?f ?p)
                (= (contents ?p ?f 1) ?K))
  :effect (= (contents ?p ?out 1) (substitute ?k "TAB" "       ")))
```

Latex. *Produce the .dvi, .aux, .log and .bbl file for the latex file ?f in pool ?p. This command has two conditional effects: (1) If the .bbl file already exists and contains the bbl information of ?f, then a cited .dvi file is produced; (2) If the .aux file already exists and contains the aux information of ?f, a labelled .aux file is produced.*

```
(defaction (latex ?f ?p ?f-dvi ?f-aux ?f-log ?f-bbl)
  :prehard (AND (file ?f ?p)
                (writable ?p)
                (readable ?f ?p)
                (= (contents ?p ?f 1) ?K))
  :effect (AND (= (contents ?p ?f-dvi 1) (latex-dvi-of ?K))
               (= (contents ?p ?f-aux 1) (latex-aux-of ?K))
               (= (contents ?p ?f-log 1) (latex-log-of ?K))
               (file ?f-dvi ?p)
               (file ?f-aux ?p)
               (file ?f-log ?p))
  :ceffect (((= (contents ?p ?f-bbl 1) (latex-bbl-of ?K))
             (cited-dvi-file ?f-dvi))
            ((= (contents ?p ?f-aux 1) (latex-aux-of ?K))
             (labelled-aux-file ?f-aux))))
```

Bibtex. *Produce the .bbl file of ?f containing the bbl information of ?f's contents.*

```
(defaction (bibtex ?f ?f-bbl ?p)
  :prehard (AND (file ?f ?p)
                (writable ?p)
                (= (contents ?p ?f 1) ?K))
  :effect (and (= (contents ?p ?f-bbl 1) (latex-bbl-of ?K))
               (file ?f-bbl ?p)))
```

Print. *Print the contents of file ?f on printer ?printer.*

```
(defaction (lp ?f ?p ?printer)
  :prehard (AND (file ?f ?p) (readable ?f ?p) (= (contents ?p ?f 1) ?k))
  :effect (printed ?printer ?k))
```

List-files. *List the contents of pool ?p into file ?out.*

```
(defaction (ls ?p ?out)
  :prehard (and (readable ?p)
                (= (files-of ?p) ?file-list)
                (in-directory ?d))
  :effect (= (contents ?out ?d 1) ?file-list))
```

Create-directory. *Create a subdirectory ?p of directory ?d.*

```
(defaction (mkdir ?p ?d)
  :prehard (writable ?d)
  :effect (AND (pool ?p ?d) (writable ?p))
  :ceffect (((readable ?d)
             (readable ?p))
            ((executable ?d)
             (executable ?p))
            ((not (readable ?d) )
             (not (readable ?p)))
            ((not (executable ?d) )
             (not (executable ?p)))))
```

Help. *Produce the documentation for command ?operator into file ?out.*

```
(defaction (man ?operator ?out)
  :prehard (file ?out ?p)
  :effect (= (contents ?p ?out 1) (documentation ?operator)))
```

Remove the subdirectory ?p of ?d.

```
(defaction (rmdir ?p ?d)
  :prehard (and (pool ?p ?d) (writable ?d) (empty ?p))
  :effect (not (pool ?p ?d)))
```

Spell. *Check the spelling of the contents of file ?f in pool ?p. If the file is correctly spelt, set a flag ?out to "F", otherwise set ?out to "T".*

```
(defaction (spell ?f ?p ?out)
  :prehard (and (file ?f ?p) (readable ?f))
  :ceffect (((not (correctly-spelt ?f)) (= ?out "F"))
            ((correctly-spelt ?f) (= ?out "T"))))
```

Substitute. *Replace all occurrences of a regular expression ?pat1 by ?pat2 in file ?f of pool ?p.*

```
(defaction (sed ?f ?p ?pat1 ?pat2)
  :prehard (and (file ?f ?p)
                (= (contents ?p ?f 1) ?k)
                (writable ?f ?p))
  :effect (= (contents ?p ?f 1) (substitute ?k ?pat1 ?pat2)))
```

Sort. *Sort the contents of file ?f of pool ?p, into file ?out. The sorting order is implicit.*

```
(defaction (sort ?f ?p ?out)
  :prehard (and (file ?f ?p)
                (readable ?f ?p)
                (= (contents ?p ?f 1) ?k))
  :effect (= (contents ?p ?out 1) (sorted ?k)))
```

Edit. *Edit a file ?f in pool ?p. The effect of the edit action is simply to modify the file.*

```
(defaction (edit ?f ?p)
  :prehard (and (writable ?p)
                (writable ?f ?p)
                (= (contents ?f ?p 1) c))
  :effect (and (not (= (contents ?f ?p 1) c))
               (modified ?f ?p)))
```

B Representative Sample of Processes

The following is a representative sample of the problems for which our algorithm was used to synthesize process programs. Except the first program, the others were obtained from the collection of shell scripts used for the empirical validation of our approach.

This is a variation of the `forced-check-in` *example in the paper, where the initial conditions are omitted. The necessary initial conditions are automatically identified during the explanation stage.*

```
(definput forced-check-in
 :initial-state nil
 :history '((move F1 F2 P1 P1)
            (check-out u1 f1 ps p1 m)
            (lock u1 f1 ps p1 m)
            (move F2 F1 P1 P1)
            (check-in U1 F1 P1 Ps))
 :goal '((= (contents P1 F1 1) (contents Ps F1 (+ M 1)))
         (= (contents P1 F1 1) C)
         (= (+ m 1) (last-version ps f1))))
```

Generalized process program produced by the system:

```
Initial State:-

Program:-
     (MOVE ?F1-37 ?F1-56 ?P1-23 ?P1-23)
     (CHECK-OUT ?U-64 ?F1-37 ?PS-36 ?P1-23 ?M-34)
     (LOCK ?U-52 ?F1-37 ?PS-36 ?P1-23 ?M-34)
     (MOVE ?F1-56 ?F1-37 ?P1-23 ?P1-23)
     (CHECK-IN ?U-52 ?F1-37 ?P1-23 ?PS-36)

Goal:-
     (= (CONTENTS ?P1-23 ?F1-37 1) (CONTENTS ?PS-36 ?F1-37 (+ ?M-34 1)))
     (= (CONTENTS ?P1-23 ?F1-37 1) ?C-33)
     (= (+ ?M-34 1) (LAST-VERSION ?PS-36 ?F1-37))

Provided the following assumptions are valid:-
     (WRITABLE ?P1-23)
     (READABLE ?F1-37 ?P1-23)
     (WRITABLE ?F1-37 ?P1-23)
     (FILE ?F1-37 ?P1-23)
     (FORALL (?U) (NOT (LOCKED ?U ?F1-37 ?PS-36 ?M-34)))
     (= (CONTENTS ?P1-23 ?F1-37 1) ?C-33)
     [Soft] (NOT (FILE ?F1-56 ?p1-23))
```

Use of a sequence of latex and bibtex commands to create a cited and labelled .dvi and .aux file. The system detects the redundancy of one latex command.

```
(definput run-latex
 :initial-state '((file myfile dir)
                  (= (contents dir myfile 1) K))
 :history '((latex myfile dir myfile-dvi myfile-aux myfile-log myfile-bbl)
            (latex myfile dir myfile-dvi myfile-aux myfile-log myfile-bbl)
            (bibtex myfile myfile-bbl dir)
            (latex myfile dir myfile-dvi myfile-aux myfile-log myfile-bbl))
 :goal '((= (contents dir myfile-dvi 1) (latex-dvi-of K))
         (= (contents dir myfile-aux 1) (latex-aux-of K))
         (= (contents dir myfile-log 1) (latex-log-of K))
         (= (contents dir myfile-bbl 1) (latex-bbl-of K))
         (cited-dvi-file myfile-dvi)
         (labelled-aux-file myfile-aux)))
```

Generalized process program produced by the system:

```
Initial State:-
     (FILE ?F-128 ?DIR-100)
     (= (CONTENTS ?DIR-100 ?F-128 1) ?K-103)

Program:-
     (LATEX ?F-128 ?DIR-100 ?F-DVI-130 ?MYFILE-AUX-109 ?F-LOG-132 ?F-BBL-133)
     (BIBTEX ?F-128 ?F-BBL-124 ?DIR-100)
     (LATEX ?F-128 ?DIR-100 ?MYFILE-DVI-108 ?MYFILE-AUX-109 ?MYFILE-LOG-101 ?F-BBL-124)

Goal:-
     (= (CONTENTS ?DIR-100 ?MYFILE-DVI-108 1) (LATEX-DVI-OF ?K-103))
     (= (CONTENTS ?DIR-100 ?MYFILE-AUX-109 1) (LATEX-AUX-OF ?K-103))
     (= (CONTENTS ?DIR-100 ?MYFILE-LOG-101 1) (LATEX-LOG-OF ?K-103))
     (= (CONTENTS ?DIR-100 ?F-BBL-124 1) (LATEX-BBL-OF ?K-103))
     (CITED-DVI-FILE ?MYFILE-DVI-108)
     (LABELLED-AUX-FILE ?MYFILE-AUX-109)

Provided the following assumptions are valid:-
     (WRITABLE ?DIR-100)
     (READABLE ?F-128 ?DIR-100)
```

Copy a non-readable file into another (readable) file. The file that is copied should remain non-readable at the end. The system identifies an important condition that was implicit in the shell script: the directory should be readable and writable.

```
(definput copy-non-readable
 :initial-state '((file foo dir)
                  (not (readable foo dir))
                  (= (contents dir foo 1) C))
 :history '((cat foo dir out1)
            (change-mode+r foo dir)
            (cat foo dir out2)
            (change-mode-r foo dir))
 :goal '((= (contents dir out2 1) C)
         (not (readable foo dir))))
```

Generalized process program produced by the system:

```
Initial State:-
     (FILE ?FOO-220 ?DIR-221)
     (NOT (READABLE ?FOO-224 ?DIR-225))
     (= (CONTENTS ?DIR-221 ?FOO-220 1) ?C-219)

Program:-
     (CHANGE-MODE+R ?FOO-220 ?DIR-221)
     (CAT ?FOO-220 ?DIR-221 ?OUT2-217)
     (CHANGE-MODE-R ?FOO-220 ?DIR-221)

Goal:-
     (= (CONTENTS ?DIR-221 ?OUT2-217 1) ?C-219)
     (NOT (READABLE ?FOO-220 ?DIR-221))

Provided the following assumptions are valid:-
     (WRITABLE ?DIR-221)
     (READABLE ?DIR-221)
     (WRITABLE ?OUT2-217 ?DIR-221)
```

Replace all TAB character by spaces, sort a file, and print it out. Note that the goal can be stated very concisely using nested expressions.

```
(definput expand-and-print
 :initial-state '((= (contents dir phones 1) P)
                  (readable out2 dir)
                  (readable out1 dir))
 :history '((sort phones dir out1)
            (expand out1 dir out2)
            (cat out2 dir pr-phones)
            (lp pr-phones dir printer))
 :goal '((= (contents dir pr-phones 1) (substitute (sorted P) "TAB" "      "))
         (printed printer (substitute (sorted P) "TAB" "      ")))
)
```

Generalized process program produced by the system:

```
Initial State:-
     (= (CONTENTS ?DIR-235 ?PHONES-243 1) ?P-253)
     (READABLE ?OUT2-237 ?DIR-235)
     (READABLE ?OUT1-240 ?DIR-235)

Program:-
     (SORT ?PHONES-243 ?DIR-235 ?OUT1-240)
     (EXPAND ?OUT1-240 ?DIR-235 ?OUT2-237)
     (CAT ?OUT2-237 ?DIR-235 ?PR-PHONES-234)
     (LP ?PR-PHONES-234 ?DIR-235 ?PRINTER-252)

Goal:-
     (= (CONTENTS ?DIR-235 ?PR-PHONES-234 1) (SUBSTITUTE (SORTED ?P-253) "TAB" "      "))
     (PRINTED ?PRINTER-252 (SUBSTITUTE (SORTED ?P-253) "TAB" "      "))

Provided the following assumptions are valid:-
     (FILE ?PHONES-243 ?DIR-235)
     (READABLE ?PHONES-243 ?DIR-235)
     (FILE ?OUT1-240 ?DIR-235)
     (FILE ?OUT2-237 ?DIR-235)
     (READABLE ?DIR-235)
     (WRITABLE ?PR-PHONES-234 ?DIR-235)
     (FILE ?PR-PHONES-234)
     (WRITABLE ?OUT1-240 ?DIR-235)
     (WRITABLE ?OUT2-237 ?DIR-235)
```

Produce three named files that contain the .dvi, .aux and .log information of a latex file.

```
(definput copy-and-run-latex
 :initial-state '((= (contents dir a 1) K))
 :history '((copy a lat-body dir dir)
            (latex lat-body dir lat-dvi lat-aux lat-log lat-bbl)
            (copy lat-dvi b dir dir)
            (move lat-aux c dir dir)
            (move lat-log d dir dir)
            (remove-file lat-body dir))
 :goal '((file b dir)
         (= (contents dir b 1) (latex-dvi-of K))
         (file c dir)
         (= (contents dir c 1) (latex-aux-of K))
         (file d dir)
         (= (contents dir d 1) (latex-log-of K))
         (not (file lat-body dir))))
```

Generalized process program produced by the system:

```
Initial State:-
     (= (CONTENTS ?DIR-320 ?F1-297 1) ?K-318)

Program:-
     (COPY ?F1-297 ?LAT-BODY-319 ?P1-299 ?DIR-320)
     (LATEX ?LAT-BODY-319 ?DIR-320 ?F1-287 ?F1-284 ?F1-281 ?F-BBL-296)
     (COPY ?F1-287 ?B-304 ?P1-289 ?DIR-320)
     (MOVE ?F1-284 ?C-310 ?DIR-320 ?DIR-320)
     (MOVE ?F1-281 ?D-316 ?DIR-320 ?DIR-320)
     (REMOVE-FILE ?LAT-BODY-319 ?DIR-320)

Goal:-
     (FILE ?B-304 ?DIR-320)
     (= (CONTENTS ?DIR-320 ?B-304 1) (LATEX-DVI-OF ?K-318))
     (FILE ?C-310 ?DIR-320)
     (= (CONTENTS ?DIR-320 ?C-310 1) (LATEX-AUX-OF ?K-318))
     (FILE ?D-316 ?DIR-320)
     (= (CONTENTS ?DIR-320 ?D-316 1) (LATEX-LOG-OF ?K-318))
     (NOT (FILE ?LAT-BODY-319 ?DIR-320))

Provided the following assumptions are valid:-
     (FILE ?F1-297 ?DIR-320)
     (READABLE ?F1-297 ?DIR-320)
     (READABLE ?DIR-320)
     (WRITABLE ?DIR-320)
     (READABLE ?F1-287 ?DIR-320)
     (READABLE ?F1-284 ?DIR-320)
     (WRITABLE ?F1-284 ?DIR-320)
     (READABLE ?F1-281 ?DIR-320)
     (WRITABLE ?F1-281 ?DIR-320)
     (WRITABLE ?LAT-BODY-319 ?DIR-320)
     [Soft] (NOT (FILE ?LAT-BODY-319 ?DIR-320))
     [Soft] (NOT (FILE ?B-304 ?DIR-320))
     [Soft] (NOT (FILE ?C-310 ?DIR-320))
     [Soft] (NOT (FILE ?D-316 ?DIR-320))
```

C References

[1] Sanjay Bhansali and Mehdi T. Harandi. Synthesis of unix programs using derivational analogy. *Machine Learning*, 10(2):7–55, 1993.

[2] A. W. Biermann and R. Krishnaswamy. Constructing programs from example computations. *IEEE Transactions on Software Engineering*, 2(3):141–153, September 1976.

[3] Brad Blumenthal. Empirical comparisons of some design replay algorithms. In *Proceedings Eighth National Conference on Artificial Intelligence*, pages 902–907, Boston, MA, August 1990. MIT Press.

[4] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

[5] Jeff Conklin and Michael L. Begeman. gIBIS: A Hypertext Tool for Exploratory Policy Discussion. In *Proc. of the 2nd Computer Supported Cooperative Work Conference*, pages 140–152, Portland, Oregon, 1988.

[6] Allen Cypher. Eager: Programming Repetitive Tasks by Example. In *Proceedings of SIGCHI Conference*, 1991.

[7] Gerald Dejong and Raymond Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1:145–176, 1986.

[8] R. E. Fikes and N. J. Nilsson. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:198–208, 1971.

[9] P. K. Garg and S. Bhansali. Process Programming by Hindsight. In *Proceedings of the* 14^{th} *International Conference on Software Engineering*, May 1992.

[10] Mehdi T. Harandi and Sanjay Bhansali. Program derivation using analogy. In *Proceedings of the Eleventh International Conference on Artificial Intelligence*, pages 389–394, Detroit, MI, August 1989. Morgan Kaufmann.

[11] Angela K. Hickman and Marsha C. Lovett. Partial match and search control via internal analogy. In *Proceedings of the Thirteenth Annual Conference of the Cognitive Science Society*, Chicago, IL, 1991. Lawrence Erlbaum.

[12] Karen E. Huff and Victor R. Lesser. A Plan-Based Intelligent Assistant that Supports the Software Development Process. In Peter Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 97–106, Boston, Massachusetts, 1988.

[13] M.N. Huhns and R.D. Acosta. Argo: An analogical reasoning system for solving design problems. Technical Report AI/ CAD-092-87, MCC, Microelectronics and Computer Technology Corporation, Austin, TX, 1987.

[14] Gail E. Kaiser, Peter H. Feiler, and Steven S. Popovich. Intelligent Assistance for Software Development and Maintenance. *IEEE Software*, pages 40–49, May 1988.

[15] M. M. Lehman. Process Models, Process Programs, Programming Support. In *Proceedings of the* 9^{th} *International Conference on Software Engineering*, pages 14–16, April 1987. Response to the keynote address, this conference, by Leon Osterweil.

[16] P. Mi, M.-J. Lee, and W. Scacchi. A knowledge-based software process library for process-driven software development. In *Proceedings of the 7th Knowledge-Based Software Engineering Conference*, pages 122–131, McLean, VA, 1992.

[17] Tom M. Mitchell, Richard M. Keller, and Smadar T. Kedar-Cabelli. Explanation-based learning: A unifying view. *Machine Learning*, 1:47–80, 1986.

[18] Jack Mostow, Michael Barley, and Timothy Weinrich. Automated reuse of design plans. *International Journal for Artificial Intelligence and Engineering*, 4(4):181–196, October 1989.

[19] Jack Mostow and Greg Fisher. Replaying transformational derivations of heuristic search algorithms in DIOGENES. In *Proceedings of the Case-based Reasoning Workshop*, pages 94–99, Pensacola Beach, Florida, May 1989.

[20] N. J. Nilsson. *Principles of Artificial Intelligence.* Tioga, Palo Alto, CA, 1980.

[21] Alberto M. Segre. *Machine Learning of Robot Assembly Plans.* Kluwer Academic Publishers, Norwell, Massachusetts, 1988.

[22] L. I. Steinberg and T. M. Mitchell. The redesign system: a knowledge-based approach to VLSI CAD. *IEEE Design & Test*, 2:45–54, 1985.

[23] Stanley M. Sutton Jr., Dennis Heimbinger, and Leon J. Osterweil. Language Constructs for Managing Change in Process-Centered Environments. In Richard N. Taylor, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 206–217, Irvine, California, December 1990.

[24] M. Suzuki and T. Katayama. Meta-operations in the process model HFSP for the dynamics and flexibility of software processes. In *Proceedings of the First International Conference on the Software Process*, Washington, D.C., 1991.

[25] R. Waldinger. Achieving several goals simultaneously. In *Machine Intelligence 8*, pages 94–138. Halstead and Wiley, New York, 1977.

[26] D. S. Wile. Program developments: formal explanations of implementations. *Communications of the ACM*, 26(11):902–911, 1983.

[27] D. Wilkins, W. Clancey, and B. Buchanan. ODYSSEUS: a learning apprentice. In *Proceedings of the International Machine Learning Workshop*, pages 221–223, Skytop, PA, 1985.