Rapid Goal-Oriented Automated Software Testing using MEA-Graph Planning

Manish Gupta, Farokh Bastani, Latifur Khan, and I-Ling Yen

Department of Computer Science

University of Texas at Dallas, TX 75083-0688

Email: [*manishg, bastani, lkhan, ilyen*]*@utdallas.edu*

Abstract

With the rapid growth in the development of sophisticated modern software applications, such as multimodal distributed systems, the complexity of the software development process has increased enormously, posing an urgent need for the automation of some of the more time-consuming aspects of the development process. One of the key stages in the software development process is system testing. In this paper, we evaluate the potential application of AI planning techniques in automating software testing. The key contributions of this paper include the following: (1) A formal model of software systems from the perspective of software testing that is applicable to important classes of systems and is amenable to automation using AI planning methods. (2) The design of a framework for an automated planning system (APS) for applying AI planning techniques for testing software systems. (3) Assessment of the test automation framework and a specific AI Planning algorithm, namely, MEA-Graphplan (Means-Ends Analysis Graphplan), algorithm to automatically generate test data.

Key Words: AI Planning, planning graph, MEA-Graphplan, automated software testing.

1. Introduction

During system or integration testing of multimodal distributed systems, it is not only necessary to understand the properties of each of the subsystem and identify the possible interactions and conflicts between subsystems, but it is also required to test the safety, security, and reliability of the system in specific states. The test engineer needs to test the system in states that are closer to forbidden regions, to see if any state transitions will cause the system to enter an unsafe state [YBM02]. To accomplish this, the test engineer needs to generate test cases manually to check whether the system reaches an unsafe state. Manual test data generation can consume a large amount of time and effort, and may not guarantee that the system will never reach the specified unsafe state.

Automated test data generation can be used to generate test data (a sequence of state transitions) that take the system from the current state to some desired state [MH93, MMW94, MSD00]. A variety of automated testing tools currently exist but most of these tools cannot ensure that the generated test data will take the system to the desired state. AI planning techniques seem to be quite promising in this field because of their emphasis on *goals*, i.e., sequences of actions (e.g., plans or test data) are generated specifically to fulfill some purpose. Some of the AI planning techniques, including plan-graph planning [BF97], plan-space planning [PW92], HTN planning [NCLM99, EHN94], and temporal-logic planning [BA01, BK00, BK96], can be potential planning techniques for automating the testing process.

Among these planning techniques, Blum and Furst's Graphplan algorithm [BF97] seems to be a promising recent development. Graphplan is a simple, elegant algorithm based on a technique called Planning Graph Analysis that yields an extremely speedy planner that, in many cases, is orders of magnitude faster than the total-order planner Prodigy [VCPB95] and the partial-order planner UCPOP [PW92]. But in the basic Graphplan algorithm, during the *graph expansion* phase, the planning graph may contain many of the actions that may be irrelevant to the goal at hand. Thus, the graph expansion algorithm is oblivious to the goal of the planning problem and, as a result, during system testing of complex systems, such as multimodal distributed systems, there might be a higher probability of state-space explosion during the graph expansion phase of the planning. MEA-Graphplan [KPL97] extends the basic Graphplan algorithm by adapting means-ends analysis [M96] to Graphplan, which makes the *graph expansion* phase goal-oriented. MEA-Graphplan involves first growing the planning graph in the backward direction by *regressing* goals over actions, and then using the resulting regression-matching graph as guidance for the standard Graphplan algorithm.

In this paper, we propose a test automation framework for applying AI planning techniques for testing software systems and using a comprehensive example to describe how MEA-Graphplan automatically generates test data for the software system. The primary contributions in this paper are as follows:

1. A formal model of software systems is presented from the perspective of software testing that is applicable to important classes of systems and is amenable to

automation using AI planning methods. The formal model of software systems includes a model of the system, a model of the actions, a model of the observations, and a model of the specification objectives of the system. We also define a software system in terms of a *state transition system* ? whose description acts as input parameters to the planning system.

2. The design of a framework for an automated planning system (APS) for applying AI planning techniques for testing software systems. Our proposed framework, APS, consists of a *Planning Domain Generator* that maps software parameters to planning parameters, and an *AI Planner* that generates a plan or sequence of actions for a specific planning problem. We also formalize definitions required for defining the planning problem for software systems, and certain restrictive assumptions for our APS.
3. Assessment of the test automation framework and a specific AI Planning algorithm, namely, MEA-Graphplan algorithm to automatically generate test data. A comprehensive example describes how we derive a *state transition system* ? for the *List* ADT, develop planning operators for the specified planning domain, and apply the MEA-Graphplan to generate an *optimized* planning graph and perform solution extraction for a planning problem.

The rest of this paper is organized as follows: In Section 2, we briefly review various AI planning techniques. In Section 3, we formally present the MEA-Graphplan algorithm and explain it detail. In Section 4, we present the conceptual model of a software system from the perspective of testing. In Section 5, we propose the automated planning system (APS) framework and provide a comprehensive example for *List* ADT. In Section 6, we briefly review the related work currently going on in this field. In Section 7, we summarize the paper and identify some future research directions.

2. Review of AI Planning Techniques

A basic *planning problem* is a triple $P = (O, s_0, g)$, where O is a collection of operators, s_0 is a state (the initial state), and g is a set of literals (the goal formula). A *plan* is any sequence of actions $\langle a_1, a_2, \ldots, a_n \rangle$ such that each a_i is an instance of an operator in O.

Nearly all AI Planning procedures are search procedures. Different planning procedures have different search spaces.

The Graphplan algorithm [BF97] alternates between two phases, namely, *graph expansion* and *solution extraction*. In the graph expansion phase, the planning graph is extended forward in time until it has achieved a necessary (but perhaps insufficient) condition for plan existence. The solution extraction phase then performs a backward-chaining search on the graph, looking for a valid plan that can satisfy the goals. If no plan is found then the cycle repeats by further expanding the planning graph. The planning graph generated is a *directed*, *leveled* graph with two kinds of nodes, i.e., *proposition* nodes and *action* nodes, arranged into levels as shown in Figure 1. *Even*-numbered levels contain proposition nodes (i.e., ground literals), *odd*-numbered levels contain action nodes (i.e., action instances) whose preconditions are present (and are mutually consistent) at the previous level, and the *zeroth*-level of the planning graph consists of proposition nodes representing the initial conditions. Edges connect proposition nodes to the action nodes (at the next level) whose preconditions mention those propositions, and additional edges connect action nodes to subsequent propositions made true by the actions' effects as shown in Figure 1. Actions that do *nothing* to a proposition are called *maintenance actions* that encode persistence.

The planning graph constructed during the planning process makes the mutual exclusion (*mutex*) relation among nodes at the same level explicitly available. Also, a valid plan found during the solution extraction phase is a planning-graph where actions at the same level are not *mutex*, each action's preconditions are made true by the plan, and all the goals are satisfied. If no plans are found, then the termination condition for Graphplan states that when two adjacent proposition levels of the forward planning-graph are identical, i.e., they contain the same set of propositions and have the same exclusivity relations, then the planning-graph has *leveled off* and the algorithm terminates with a "No-Plan Exists" output signal [BF97]. Graphplan planning is both *sound* and *complete.*

In plan-space planning [W94, PW92, MR91], each node of the search space is a *partial plan* having a set of partially-instantiated actions and a set of constraints. It makes more and more refinements until we have a solution where the solution is a node (not a *path*). It

is also called *partial order planning* or *least commitment planning*. It is both *sound* and *complete.*

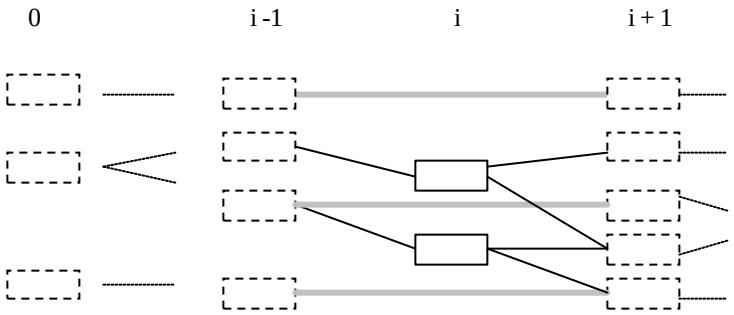

Figure 1. The Planning graph with action nodes represented by solid-squares, proposition nodes represented by dashed-squares, horizontal grey lines between proposition nodes representing the maintenance actions that encode persistence, and solid lines from proposition nodes to action nodes and vice versa represents actions' preconditions and effects respectively.

HTN planning [Y90, NCLM99, EHN94, BW94] is a type of problem reduction involving decomposition of tasks into subtasks. Each task is associated with a set of methods. Each method will have constraints associated with it. It resolves interactions and, if necessary, backtracks and tries other decompositions during plan generation. In HTN planning, plans may interleave subtasks of different tasks. If the precondition-inference procedure in HTN planning is sound and complete, then HTN planning is also *sound* and *complete.*

Among these planning techniques, the Graphplan algorithm seems to be an appropriate planner for automated test data generation since the planning graph constructed during the planning process makes useful constraints, such as interactions and conflicts among subsystems, explicitly available that might provide a better understanding of the properties of the subsystems. It also yields an extremely speedy planner that, in many cases, is orders of magnitude faster than total-order and partial-order planners.

3. MEA-GraphPlan Planning

In the basic Graphplan algorithm, during the *graph expansion* phase, the planning graph with *n*-levels contains only those actions that could possibly be executed in the initial state or in a world reachable from the initial state. But many of the actions in the planning

graph may be irrelevant to the goal at hand. Thus, the graph expansion algorithm is not informed of the goal of the planning problem [W99, KPL97, NDK97, M96, G00] and, as a result, during system testing of complex systems, such as multimodal distributed systems, there might be a higher probability of state-space explosion during the graph expansion phase of the planning.

MEA-Graphplan adapts means-ends analysis [M96] to Graphplan in order to make it goal-oriented. MEA-Graphplan [KPL97] involves first growing the planning graph in the backward direction by *regressing* goals over actions, and then using the resulting regression-matching graph as a guidance for the standard Graphplan algorithm. More specifically, regression-matching graph shows all actions that are relevant at each level of the forward planning-graph. Thus, we can now run the standard Graphplan algorithm making it consider only those actions that are present at the corresponding level of the regression-matching graph. The MEA-Graphplan algorithm is shown in Figure 2.

During regression-matching graph generation, we consider only those sub-paths that can reach the initial condition from the sub goal. Also, while determining the relevant action-set we always include the "no-action" operation. In Section 5.5, we will illustrate how the MEA-Graphplan algorithm can be applied to generate an *optimized* planning graph and perform solution extraction for a planning problem.

Loop
- Generate Regression-matching graph:
  - Construct the "regression-matching graph" by *regressing* each goal over actions, till it reaches *initial condition*.
- Determine relevant action-set:
  - Using "regression-matching graph" determine the relevant action-sets for each of the corresponding action-level.
- Graph expansion:
  - Construct the "forward-planning graph" considering only the specified actions at the corresponding action-levels.
- Solution extraction:
  - Do backward-chaining search, looking for a correct plan
  - If we find one, then return it,
  - Else, the last *proposition-level* represents the set of new *initial conditions*.

Repeat

Figure 2. MEA-Graphplan algorithm.

4. A Testing-Oriented Model of Software Systems

In order to apply AI planning techniques for testing software systems, the conceptual ingredients of a software system should include a model of the system (possible states), a model of how the system can be changed (effects of actions), a model of observation of the system, and a specification of the objectives (global constraints, forbidden regions in the system). If we define our software system as a pure Abstract Data Type (ADT) or as a process-control system (PC), then a conceptual testing-oriented model of the system can be represented as shown in Figure 3.

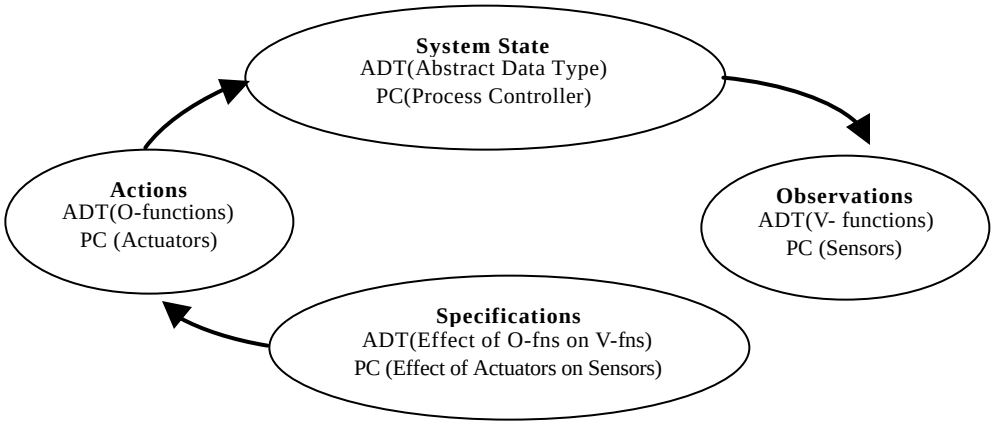

Figure 3. Conceptual Testing-Oriented Model for Software Systems.

O-functions (operation-performing functions) and *Actuators* represent methods in each class of the software system that causes transitions in the state space. Also, *V-functions* (value-returning functions) and *Sensors* represent methods in each class of the software system that returns some information about the current value of the state space. Software system specification represents the standard pre-/post-conditions or algebraic specifications for methods in each class of the software system.

Let us define our software system as a *state transition system* ? :

? = {*S, A, T, C, f*} where,

$S = \{s_1, s_2, \ldots, s_m\}$ is a set of states represented using *V*-functins/Sensors,

$A = \{a_1, a_2, \ldots, a_n\}$ is a set of actions represented using *O*-functions/Actuators,

T = Testing requirements for the software system,

C = Global constraints (corresponding to forbidden regions in the state space),

$f : S \times A \rightarrow 2^S$ is a state transition function.

The description of the state transition system Σ acts as input parameters to the planning system as discussed in the next section.

5. Automated Planning System

In order to build a general framework for applying AI planning techniques for testing software systems, we need to understand what are the key input requirements of the planning system, what are the general considerations needed for defining a planning problem for a software system, and what are the outputs that the planning system generates. We propose an automated planning system (APS) framework consisting of two components, namely, *Planning Domain Generator* and *AI Planner* as shown in Figure 4.

5.1.1 Planning Domain Generator

The component *Planning Domain Generator* maps software parameters (elicited from the state transition system Σ) to the planning parameters that are passed as inputs to the component *AI Planner*.

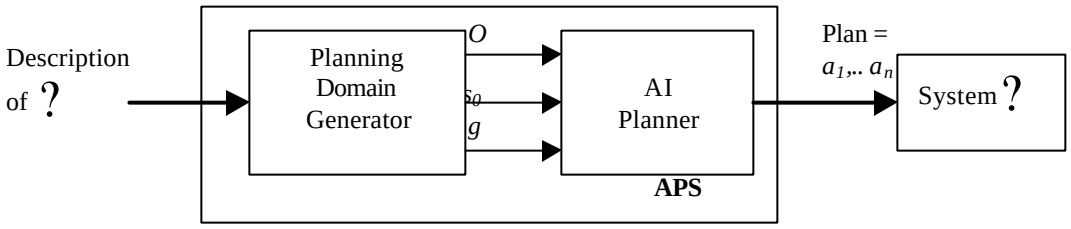

Figure 4. Automated Planning System.

We define the planning parameters as $\{O, s_0, g\}$, where O is a collection of operators, s_0 is the initial state, and g is the goal state for a specific planning problem.

We formalize some of the definitions required for defining the planning problem for the software system.

?? Initial state of a software system,

Definition 1.1: *For any planning problem, the initial state s_0 is represented using predicates over V-functions for ADTs or using predicates over Sensors for process-control systems.*

Definition 1.2: *We represent a software system as a collection of finite-state machines where each finite-state models represents the current state of a specific sub-module within it.*

?? Operator set of a software system,

Definition 2.1: *For ADTs the operator set O is defined using each relevant O-function as an operator in the planning domain. Similarly, for process-control systems the operator set is defined using relevant Actuators for the system.*

Definition 2.2: *While defining an operator's preconditions and effects, the algebraic specification of ADTs are used. Similarly, for process-control systems, an Actuator's pre-/post-conditions are used.*

?? Goal state of a software system,

Definition 3.1: *For any planning problem, the goal state g is represented using the testing requirements (T) and the global constraints (C) that need to be met during software testing.*

Our current automated planning system (APS) can be applied to test software systems with certain restrictive assumptions as discussed in the following definition.

Definition 4.1: *APS is applicable for testing a state transition system* ? *= {S, A, T, C, f} where S represents a finite set of states, A represents only controllable actions (i.e., no uncontrollable event exists), the goal state g represented using T and C are always restricted goals (i.e., g ? S), and f represents deterministic state-transition functions (i.e., no uncertainty exists).*

5.1.2 AI Planner

The component *AI Planner* takes the planning parameters $\{O, s_0, g\}$ as an input for a specific planning problem and generates a plan or sequence of actions $<a_1, a_2, \ldots, a_n>$ where each a_i is an instance of an operator in O it uses, such that the goal state is achieved. The *AI Planner* component can use any AI planning technique for generating the plan since all the AI planning techniques take the planning parameters $\{O, s_0, g\}$ as the standard input.

5.2 Example: Testing an ADT

In order to illustrate how AI planning techniques can be used to test software systems we will show how to test an abstract data type (ADT). Consider an ADT *List L* having the following methods: *create(), append(L, e, i), remove(L, i), delete(), length(L),* and *ith(L, i)*. The algebraic specification for each method in the *List* ADT is shown in Figure 5.

We define *List* ADT as a *state transition system* ? = {*S, A, T, C, f*} where,

S = set of states represented using *V*-functions: *{length(L), ith(L, i), 1 ? i ? length(L)},*

A = set of actions represented using *O*-functions: *{create(), append(L, e, i), remove(L, i), delete()},*

T =Test how *each O*-function affects the *V*-functions of the List ADT,

C = Operators do not create or destroy the main object,

The description of the state transition system ? acts as input parameters to the planning system. *APS* component *Planning Domain Generator* maps these input parameters to the planning parameters {*O*, s_0, *g*} that are passed as input to the component *AI Planner*. For our example, the component *AI Planner* uses the MEA-Graphplan technique. MEA-Graphplan extends the basic Graphplan algorithm by adapting means-ends analysis [M96] to Graphplan, which makes the *graph expansion* phase goal-oriented.

5.3 Domain Analysis

In the planning domain, the List object can be viewed as an *ordered* set of element objects where each element object is at a particular position in the List. Using simple *predicates* over the *V*-functions, we can easily define the current state of the List object.

Consider the List object shown on the left hand side of Figure 6. We can define its current state using predicates: {(*length = 4*), (*at A 1*), (*at B 2*), (*at C 3*), (*at D 4*)}, i.e. the List object length is *four*, element object A is at position 1, element object B is at position 2, and so on.

As per the algebraic specification, method *append*(*L, e, i*) increases List object length by *one*, appends object "*e*" at position "*i*" in the List, and *shifts* all the element objects at positions *j* ? *i* by *one-place* to the right. The effect of method *append*(*L, E, 3*) on the List object is shown in Figure 6. The method *remove(L, i)* decreases the length of the List object by *one*, removes the element object at position "*i*" in the List, and *shifts* all the

element objects at positions $j > i$ by *one-place* to the left. The effect of method *remove*(*L*, *3*) on the List object is shown in Figure 6.

V-functions

?? *length : : List ✍ natural*
?? *ith : : L: List x i: positive ✍ element*
![1 ? i ? L.length()] ✍ raise Bad Index

O-functions

?? *create : : ✍List*
create( ).length() = 0
?? *append : : L: List x e : element x i :natural ✍ List*
![0 ? i ? L.length()] ✍ raise Bad Index
L.append(i, e).length() =L.length() + 1
L.append(i, e).ith(j) = if j ? i ✍ L.ith(j) |
j = i + 1 ✍ e |
j > i + 1 ✍ L.ith(j – 1)
endif
?? *remove : : L: List x i :natural ✍ List*
![1 ? i ? L.length()] ✍ raise Bad Index
L.remove(i).length() = L.length() – 1
L.remove(i).ith(j) = if j < i ✍ L.ith(j) |
j ? i ✍ L.ith(j + 1)
endif
?? *delete : : List ✍*

Figure 5. Algebraic specification of List ADT.

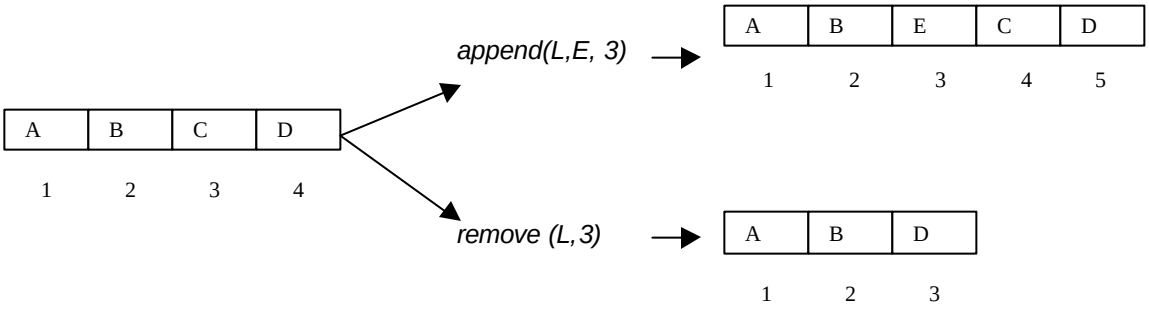

Figure 6. List Object affected by actions append(L,E,3) and remove(L,3).

5.4 Operator Definitions

For constructing the operator set *O*, each relevant *O*-function of *List* ADT is defined as an operator in the planning domain. Since the global constraint(*C*) states that the operators

do not create or destroy the main object, so we will not have operators for *O*-functions *create* and *delete*. The operators that were defined for the *List* ADT, based on its algebraic specification, are shown in Figure 7.

```
(define (operator append)
        : parameters ((element ?e) (place ?j) ( length ?len))
        : precondition (:and (less-than-equal ?j ?len))
        : effect (:and (has-length-increment ?len) (not (has-length ?len))( at ?e ?j)
          (forall (?x - location)
                              (when (greater-than-equal ?x ?j)
                                  (forall (?y - element)
                                      (when (at ?y ?x)
                                          (and (at-increment ?y ?x) (not (at ?y ?x)))))))))

(define (operator remove)
        : parameters ((place ?j) ( length ?len))
        : precondition (:and (less-than-equal ?j ?len))
        : effect (:and (has-length-decrement ?len) (not (has-length ?len))
          (forall (?x - location)
                              (when (equal-to ?x ?j)
                                  (forall (?y - element)
                                          (when (at ?y ?x)
                                                  (not (at ?y ?x)) ))))
          (forall (?x - location)
                              (when (greater-than ?x ?j)
                                  (forall (?y - element)
                                      (when (at ?y ?x)
                                          (and (at-decrement ?y ?x) (not (at ?y ?x)))))))))
```

Figure 7. List of operators in the List ADT planning problem.

Similar to STRIPS-like planning domain [FN71], we define operators that have a *name*, *parameter-list*, *preconditions*, and *effects*. Both the preconditions and effects are conjunctions of literals or propositions, and have parameters that can be instantiated to objects in the world. We define an *action* as a fully-instantiated operator.

Notice that in Figure 7, unlike a STRIPS representation in which actions are limited to *unconditional-effects*, *quantifier-free* preconditions, and effects, we are using a more expressive representation. Specifically, we have used *universal-quantified-conditional* effect that describes how an action can affect element objects at specific locations in the List. We also consider the predicate (*has-length-increment ?len*) as being equivalent-to

(*has-length* (*?len+1*)), predicate (*at-increment ?y ?x*) equivalent-to (*at ?y* (*?x+1*)), predicate (*has-length-decrement ?len*) equivalent-to (*has-length* (*?len-1*)), and predicate (*at-decrement ?y ?x*) equivalent-to (*at ?y* (*?x-1*)).

5.5 Planning Problem

Problem 1: Generate a sequence of actions to bring the List object *L* from an initial state where *{(length = 3)}* to a target state where *{(length = 4)* ***and*** *(at A 1)* ***and*** *(at D 4)}*.

By applying the MEA-Graphplan algorithm, we proceed as follows:

Step1: Generate the regression-matching graph by regressing *each* goal over actions, till it reaches *initial condition*. Figure 8(a) shows the regression-matching graph for the initial condition set: *{(length = 3)}*.

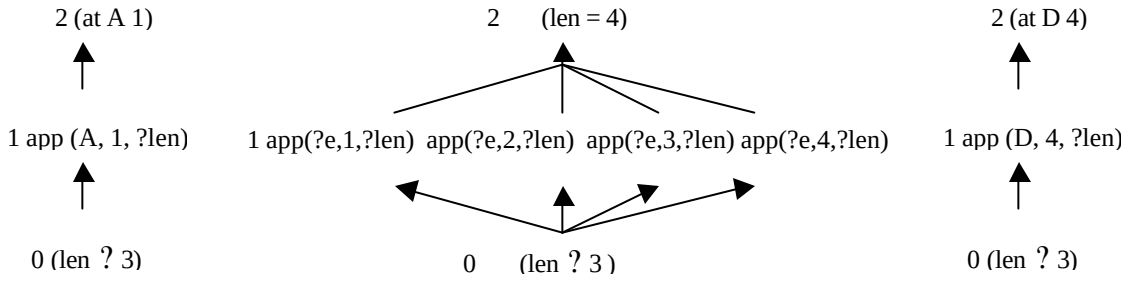

Figure 8(a). Regression-match graph with initial conditions at proposition-level 0.

Step 2: Using the regression-matching graph, determine the relevant action sets for the corresponding action-levels. Since there is only one level, so the relevant action set at level one is as follows:

Relevant_Actions_Level_1 = *{app(A,1); app(*,1); app(*,2); app(*,3); app(*,4); app(D,4); no-op}*.

Step 3: Construct the forward planning-graph by considering only the specified actions at the corresponding action-level and adding in *mutex* relations. Figure 8(b) shows the *optimized* forward planning graph constructed.

Step 4: Solution extraction fails to find a valid plan, so re-generate the regression-matching graph by considering the last proposition level as a set of *new initial-conditions*.

Proposition level 2 has the following set of *new initial conditions*: *{(at A 1), (at * 1), (at * 2), (at * 3), (at * 4), (len = 4), (at D 4), (len = 3)}*. Figure 8(c) shows the subset of the regression-matching graph for the new initial condition set.

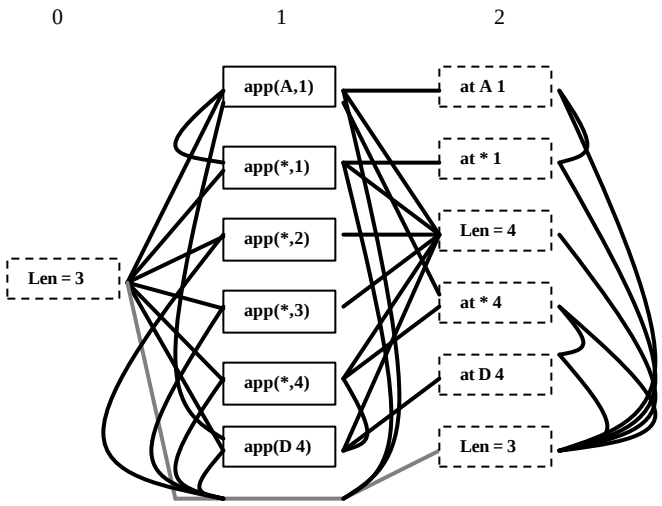

Figure 8(b). *Optimized* forward planning graph with action nodes represented by solid-squares, proposition nodes represented by dashed-squares, and horizontal lines between proposition nodes represent the maintenance actions. Thin curved lines between actions (propositions) at a single level denote mutex relations. Some of the no-ops and proposition nodes have not been specified for simplicity.

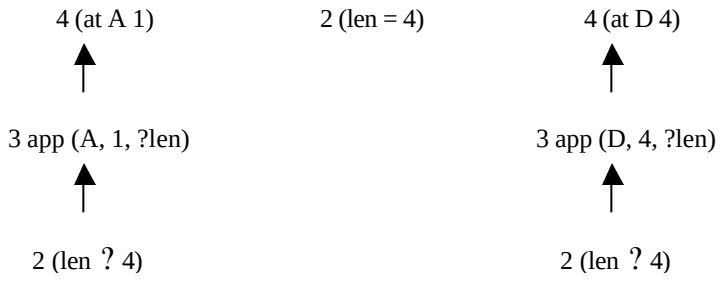

Figure 8(c). Regression-match graph subset with initial conditions at proposition-level 2.

Step 2 Repeated: Using the regression-matching graph, the relevant action set at level 3 is again:

Relevant_Actions_Level_3 = *{app(A,1); app(*,1); app(*,2); app(*,3); app(*,4); app(D,4); no-op}*.

Step 3 Repeated: Further grow the initial planning-graph as shown in Figure 8(e) till proposition level 4, considering only the relevant actions at action-level 3 and adding in *mutex* relations.

Step 4 Repeated: Solution extraction again fails to find a valid plan, so re-generate the regression-matching graph. Proposition level 4 has the following set of *new initial conditions*: *{(at A 1), (at * 1), (at * 2), (at * 3), (at * 4), (at * 5), (len = 5), (at D 4), (len = 4), (len = 3)}*. Figure 8(d) shows the subset of the regression-matching graph for the new initial condition set.

Steps 2 & 3 Repeated: Using the regression-matching graph, the relevant action set at level 5 includes *{rm(1); rm(2); rm(3); rm(4); rm(5); no-op}*. Further growing the initial planning-graph till proposition level 6, and performing the solution extraction phase of Graphplan algorithm results in a valid plan: *{app(A, 1), app(D, 4), rm(*, 5)}* shown as the dark lines in the planning graph in Figure 8(e).

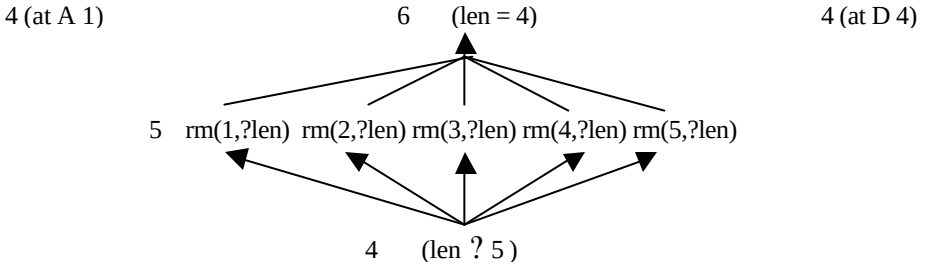

Figure 8(d). Regression-match graph subset with initial conditions at proposition-level 4.

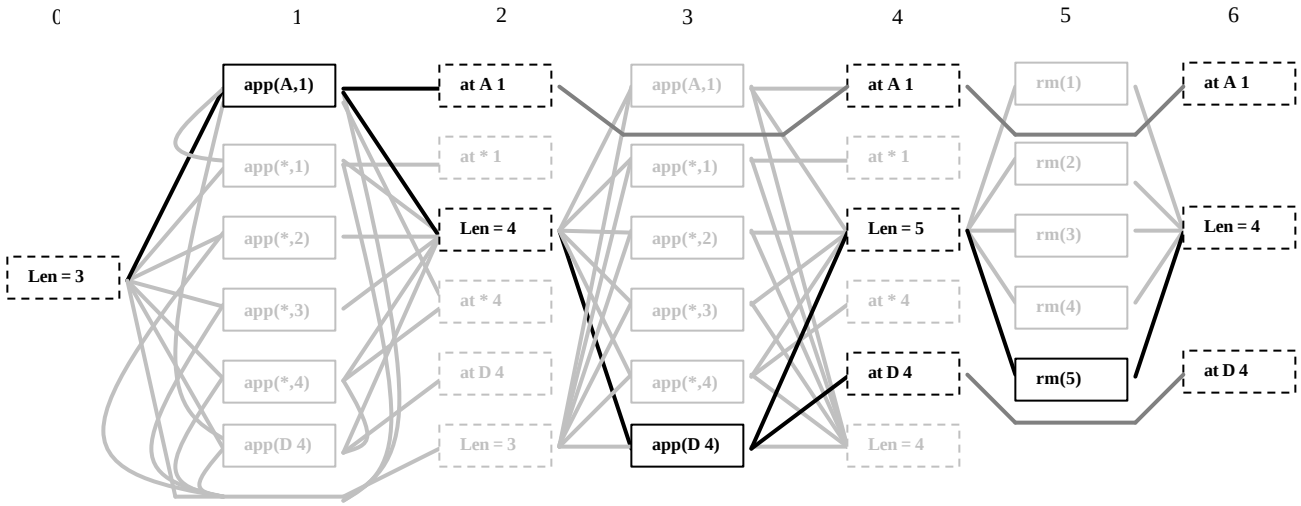

Figure 8(e). *Optimized* forward planning graph with some of the mutex relations and no-ops not shown for simplicity. The valid plan shown as the dark lines in the planning graph.

Problem 2: Generate a sequence of actions to bring List object *L* from an initial state where *{(length = 4)* ***and*** *(at A 1)* ***and*** *(at D 4)}* to a target state where *{(length = 3)* ***and*** *(at A 1)* ***and*** *(at E 3)}*.

By applying the MEA-Graphplan algorithm, we proceed as follows:

Step1: Generate the regression-matching graph by regressing *each* goal over actions, till it reaches *initial condition*. Figure 9(a) shows the regression-matching graph for the new initial condition set: *{(length = 4), (at A 1), (at D 4)}*.

Step 2: Using the regression-matching graph, determine the relevant action sets for the corresponding action-levels. Since there is only one level, so the relevant action set at level one is as follows:

Relevant_Actions_ Level_1 = *{app(E,3); rm(1); rm(2); rm(3); rm(4); no-op}*.

Step 3: Construct the forward planning-graph and add the *mutex* relations. Figure 9(b) shows the optimized forward planning graph constructed.

Step 4: Solution extraction fails to find a valid plan, so re-generate the regression-matching graph by considering the last proposition level as a set of *new initial-conditions*.

Proposition level 2 has the following set of *new initial conditions*: *{(at A 1), (len = 4), (len = 3), (len = 5), (at E 3), (at D 5), (at * 4)}*. Figure 9(c) shows the regression-matching graph for the new initial condition set.

Step 2 Repeated: Using regression-matching graphs, determine the relevant action set for the corresponding action-levels. The relevant action sets at various levels are as follows:

Relevant_Actions_Level_3 = *{rm(1); rm(2); rm(3); rm (4); rm(5); app (E, 3);no-op}*

Relevant_Actions_Level_5 = *{ rm(1); rm(2); rm(3); rm (4); no-op}*

Step 3 Repeated: Further grow the initial planning-graph as shown in Figure 9(d) till proposition level 6, considering only the specified actions at the corresponding action-

level, and adding in *mutex* relations. Now by performing the solution extraction phase of Graphplan algorithm we obtain a valid plan. The valid plan that the solution extraction finds in this example is {app(E, 3), rm(4), rm(4)} shown as the dark lines in the planning graph in Figure 9(d).

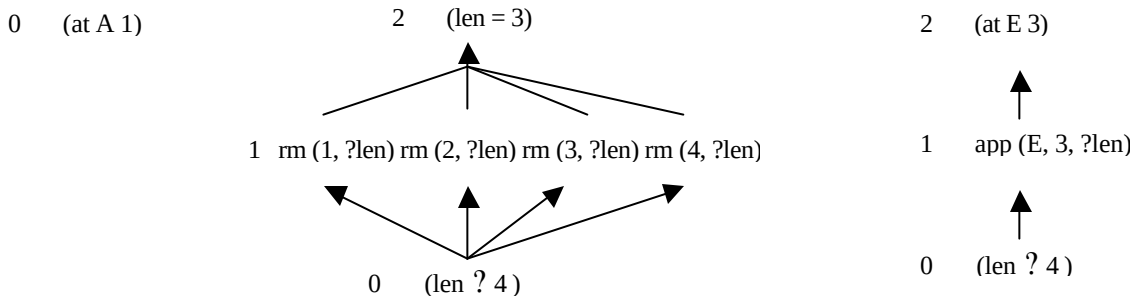

Figure 9(a). Regression-match graph with initial conditions at proposition-level 0.

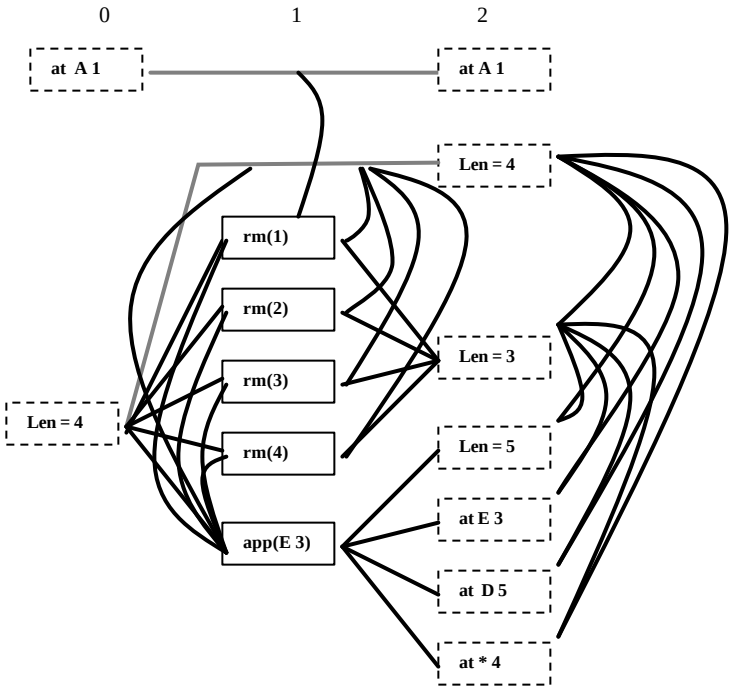

Figure 9(b). *Optimized* forward planning graph with mutex relations.

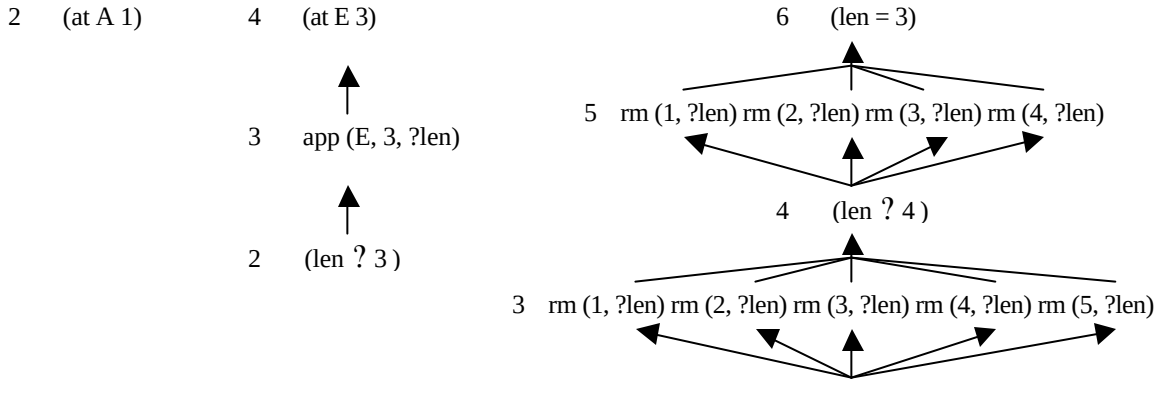

Figure 9(c). Regression-match graph subset with initial conditions at proposition-level 2.

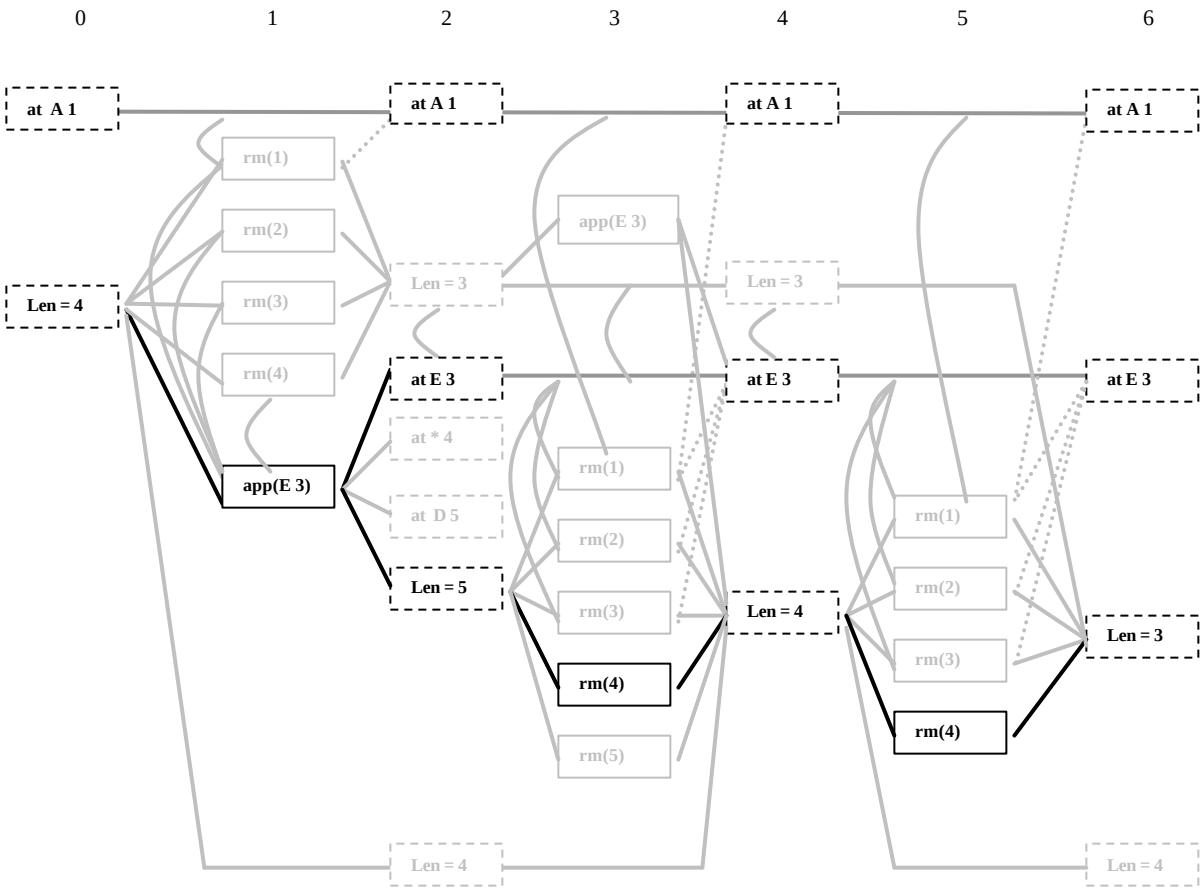

Figure 9(d). *Optimized* forward planning graph with some of the mutex relations and no-ops not shown for simplicity. The valid plan shown as the dark lines in the planning graph.

5.6 Summary

We have presented an automated planning system (APS) framework and used a comprehensive *List* ADT example to describe how the operators can be developed for the specified planning domain. We have also shown how to apply the MEA-Graphplan algorithm to generate an *optimized* planning graph and perform solution extraction for a

planning problem that might prove to be computationally more efficient and effective than the basic Graph Planning, especially for generating test cases for complex systems such as multimodal distributed systems.

6. Related Work

AI planning is attractive for software engineering processes because of its emphasis on goals and the similarity of plans to programs. AI Planning has been used for various applications within software engineering. In [FA88, A93], the authors used planning as the underlying representation for software requirements and specifications. The AI Planner automates portions of the requirements engineering processes including proposing functional specification, reviewing the specifications, and modifying the specifications to meet customer needs. In [FH92], the authors used planning in designing composite systems. The AI planner generates example plans that violate problem constraints and simulates portions of the design that helped in expediting design decision-making and evaluation. In [HL88, H92], the authors exploited the structure of plans and their ability to relate disparate goals in software engineering applications including process engineering and software adaptation. In [R92], the author represented different levels of functionality and goals in programs using a plan representation, which aided in program design and re-use.

Starting from the early-90s, some interesting work has been done in the field of automated software testing using AI based approaches. In [DBC91], the authors used rule based test generation method that encodes white box criteria and information about control and data flow of the code. In [CN94], the authors used a resolution-refutation theorem prover to determine structural test coverage and coverage feasibility. In [ZW93], the authors used knowledge base of entities, their relationships, and their refinements for refinement of the test case description. Anderson *et al.* [AMM95] used neural networks as classifiers to predict which test cases are likely to reveal faults.

In some of the most recent works on automated software testing the focus has been on using AI planning techniques because of its emphasis on goals and the similarity of plans

and test cases. In [HMM97, SMF99, MHML95], the authors use partial-order planner UCPOP for test case generation. The main reasons for using UCPOP (Universal Conditional Partial Order Planner) are that it is relatively easy to use and the domain representation is richer since it can represent goals that include universal quantifiers and it does not order the operators in the plan until necessary. One major shortcoming of partial-order planning over total-order planning is the *linkability* issue as discussed in [VB94].

In [MPS01], the authors use Interference Progression Planner (IPP), and HTN planning for test case generation. IPP [ASW98, KNHD97], a descendant of Graphplan, yields an extremely speedy planner that in many cases is several orders of magnitude faster than the total-order planner Prodigy [VCPB95] and the partial-order planner UCPOP [PW92]. HTN planning, also referred to as Hierarchical planning, seems to be quite valuable for GUI test case generation as GUIs typically have a large number of components and events, and the use of hierarchy allows the GUI to be conceptually decomposed into different levels of abstraction resulting in greater planning efficiency.

In [GBKY04], we used Graphplan and its descendent MEA-Graphplan to generate test data for software systems. We find Graphplan and its descendents to be the most appropriate planning technique for testing complex systems such as multimodal distributed systems, since the planning graph constructed during the planning process makes useful constraints (*mutual exclusion* relation between action nodes and proposition nodes) explicitly available. It also yields an extremely speedy planner. Also, MEA-Graph planning might solve the problem of state-space explosion during the graph expansion phase of the planning process. In [GBKY04], we also propose an automated planning system for applying any of the available AI planning techniques for software testing of an ADT.

7. Conclusion and Future Work

During automated software testing, AI planning techniques Graphplan and its descendents MEA-Graph planning might help us better understand the properties of the subsystems by identifying possible interactions and conflicts between subsystems using its planning graph. AI planning techniques generate a sequence of actions (e.g., plan or

test data) that guarantee that the system reaches its goal state thus allowing us to test the system in states that are closer to forbidden regions. Further, MEA-Graph planning might prove to be computationally more efficient and effective than basic Graph Planning especially for system testing for complex systems.

One immediate research direction is to evaluate the performance of the MEA-Graph planning technique in comparison with other AI planning techniques both analytically and experimentally using a comprehensive set of examples and also to show empirical results. Another research direction is to explore the potential application of some learning techniques [MCKK89, BV96, V94] in AI planning for automated software testing. Learning in AI planning can basically be categorized into learning in total-order planner [AANW01, V94, VCPB95], and learning in partial-order planner [EM96, M98, MW97]. During unit testing of subsystems, we can train the AI planner to generate plans for individual subsystems. This may help to speed up the planning process during system or integration testing. Thus, learning in AI planning may lead to more efficient and faster planning for automated software testing.

References

[AANW01] H. M. Avila, D. W. Aha, D. S. Nau, R. Weber, L. Breslow, and F. Yaman, "SiN: Integrating case-based reasoning with task decomposition", In *IJCAI-2001*, Seattle, August, 2001.

[A93] J. S. Anderson, "Automating requirements engineering using Artificial Intelligence Techniques, *PhD thesis, Dept. of Computer and Information Science*, University of Oregon, December 1993.

[AMM95] C. Anderson, A. Mayrhauser, and R. Mraz," On the use of Neural Networks to guide Software Testing Activities", In *Proc. of International Test Conference*, October 1995, Washington, DC.

[ASW98] C. Anderson, D. E. Smith and D. Weld, "Conditional effects in graphplan", In *Proc. of 4th Intl. Conf. on AI Planning Systems*, June 1998.

[BA01] F. Bacchus and M. Ady, "Planning with resources and concurrency: A forward chaining approach", *International Joint Conference on Artificial Intelligence (IJCAI-2001)*, pp. 417-424, 2001.

[BF97] A. Blum and M. Furst, "Fast planning through planning graph analysis", *Artificial Intelligence*, 90:281-300, 1997.

[BK96] F. Bacchus and F. Kabanza, "Using temporal logic to control search in a forward chaining planner", *New Directions in Planning, M. Ghallab and A. Milani (Eds.) IOS Press*, pp. 141-153, 1996.

[BK00] F. Bacchus and F. Kabanza, "Using temporal logic to express search control knowledge for planning", *Artificial Intelligence*, vol 116, 2000.

[BV96] D. Borrajo and M. M. Veloso, "Lazy incremental learning of control knowledge for efficiently obtaining quality plans", *AI Review Journal. Special Issue on Lazy Learning*, 10:1-34, 1996.

[BW94] A. Barrett and D. Weld, "Task-decomposition via plan parsing", In *Proc. of AAAI-94*, Seattle, WA, July 1994.

[CN94] J. J. Chilenski and P. H. Newcomb, "Formal specification Tools for Test Coverage Analysis", In *Proc. of Ninth Knowledge-Based Software Engineering Conference*, September 1994, Monterey, CA, pp. 59-68.

[DBC91] W. Deason, D. Brown, K. H. Chang, and J. Cross, "Rule-Based software test data generator", *IEEE Transactions on Knowledge and Data Engineering*, 3(1), March 1991, pp. 108-117.

[EHN94] K. Erol, J. Hendler, and D. S. Nau, "UMCP: A sound and complete procedure for hierarchical task-network planning", In *Proc. of the International Conference on AI Planning Systems (AIPS)*, pp. 249-254, June 1994.

[EM96] T. A. Estlin and R. J. Mooney, "Hybrid learning of search control for partial-order planning", *New Directions in AI Planning, IOS Press*, 1996, pp. 129-140.

[FA88] S. Fickas and J. Anderson, "A proposed perspective shift: Viewing specification design as a planning problem", *Department of Computer and Information* Science, *CIS-TR-88-15*, University of Oregon, Eugene, OR, November 1988.

[FH92] S. Fickas and B. R. Helm, "Knowledge representation and reasoning in the design of composite systems", *IEEE Transactions on Software Engineering*, SE-18(6), June 1992, pp. 470-482.

[FN71] R.Fikes, and N. Nilsson, "STRIPS: A new approach to the application of theorem proving to problem solving", *Artificial Intelligence*, 2(3/4), 1971.

[G00] M. Garagnani, "Extending graphplan to domain axiom planning", In *Proc. of the 19th Workshop of the UK Planning and Scheduling SIG (PLANSIG 2000)*, Milton Keynes (UK), ISSN 1368-5708, pp. 275-276, December 2000.

[GBKY04] M. Gupta, F. Bastani, L. Khan, and I. L. Yen, "Automated test data generation using MEA-Graph Planning", In *Proc. of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'04)*, Boca Raton, Florida (USA), pp. 174-182, November 2004.

[H92] K. Huff, "Software adaptation", In *Working Notes of AAAI-92 Spring Symposium on Computational Considerations in Supporting Incremental Modification and Reuse*, Stanford University, pp. 63-66, March 1992.

[HL88] K. Huff and V. Lesser, "A plan-based intelligent assistant that supports the software development process", *ACM SIGSOFT/SIGPLAN, Software Engineering Symposium on Practical Software Development Environments*, November, 1998.

[HMM97] A. Howe, A. Mayrhauser and R. Mraz "Test case generation as an AI planning problem", *Automated Software Engineering*, Vol.4, No.1, pp. 77-106, January, 1997.

[KNHD97] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos, "Extending planning graphs to an ADL subset", In *Proc. 4th European Conference on Planning*, pp. 273-285, Sept 1997.

[KPL97] R. Kambhampati, E. Paeker, and E. Lambrecht, "Understanding and extending graphplan", In *Proc. 4th European Conference on Planning*, Sept. 1997.

[M96] D. McDermott, "A heuristic estimator for means-ends analysis in planning", In *Proc. 3rd Intl. Conf. AI Planning systems*, pp. 142-149, May 1996.

[M98] H. Muñoz-Avila, "Integrating twofold case retrieval and complete decision replay in CAPlan/CbC", PhD Thesis, University of Kaiserslautern, May 1998.

[MCKK89] S. Minton, J. G. Carbonell, C. A. Knoblock, D. R. Kuokka, O. Etzioni, and Y. Gil, "Explanation-based learning: A problem-solving perspective", *J. Artificial Intelligence*, 40(1-3): 63-118, 1989.

[MH93] A. Mayrhauser, and S. C. Hines, "Automated testing support for a robot tape library", In *Proc. of the Fourth International Software Reliability Engineering Conference*, pp. 6-14, November 1993.

[MHML95] R.T Mraz, A.E Howe, A. Mayrhauser, and L. Li, "System testing with an AI planner", In *Proc. Sixth International Symposium on Software Reliability Engineering*, pp. 96 –105, Oct. 1995.

[MMW94] A. Mayrhauser, R.T. Mraz, and J. Walls, "Domain based regressing testing," In *Proc. of the International conference on Software Maintenance*, pp. 26-34, Sept 1994.

[MPS01] A. M. Memon, M. E. Pollack, and M. L. Soffa, "Hierarchical GUI test case generation using automated planning", *IEEE Transactions on Software Engineering*, Vol. 27, No.2, February 2001.

[MR91] D. McAllester and D. Rosenblitt, "Systematic nonlinear planning", In *Proc. 9th National Conference on Artificial Intelligence*, 1991.

[MSD00] A. Mayrhauser, M. Scheetz, E. Dahlman, and A.E Howe "Planner based error recovery testing", In *Proc. 11th International Symposium on Software Reliability Engineering*, pp. 186 –195, Oct. 2000.

[MW97] H. Muñoz-Avila, and F. Weberskirch, "A case study on the mergeability of cases with a partial-order planner", In *Proc. of ECP-97, Steel & R. Alami (Eds.): Recent Advances in AI Planning*, Springer, 1997.

[NCLM99] D. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila, "SHOP: Simple hierarchical ordered planner", *IJCAI-99*, pp. 968-973, 1999.

[NDK97] B.Nebel, Y. Dimopoulos, and J. Koehler, "Ignoring irrelevant facts and operators in plan generation", In *Proc. 4th European Conference on Planning*, Sept 1997.

[PW92] J. S. Penberthy and D. Weld, " UCPOP: A sound, complete, partial order planner for ADL", In *Proc. 3rd Intl. Conf. Principles of Knowledge Representation and Reasoning*, pp. 103-114, Oct. 1992.

[R92] R. S. Rist, "Plan identification and re-use in programs", In *Working Notes of AAAI-92 Spring Symposium on Computational Considerations in Supporting Incremental Modification and Reuse*, Stanford University, pp. 67-72, March 1992.

[SMF99] M. Scheetz, A. Mayrhauser, R. France, E. Dahlman, and A. Howe, "Generating test cases from OO model with an AI planning system", In *Proc. of 10th International Symposium on Software Reliability Engineering*, pp. 250-259, November 01 - 04, 1999.

[V94] M. M. Veloso, "Flexible strategy learning: Analogical replay of problem solving episodes", In *Proc. of AAAI-94, the Twelfth National Conference on Artificial Intelligence*, pp. 595-600, Seattle, WA, August 1994. AAAI Press.

[VB94] M. M. Veloso and J. Blythe, "Linkability: Examining causal link commitments in partial-order planning," In *Proc. of the Second International Conference on AI Planning Systems*, pp. 170-175, June 1994.

[VCPB95] M. M. Veloso, J. Carbonell, M. A. Perez, Daniel Borrajo, Eugene Fink, and Jim Blythe, "Integrating planning and learning: The prodigy architecture", *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1): 81-120, 1995.

[W94] D. S. Weld, "An Introduction to least-commitment planning", *AI Magazine*, 15(4), pp. 27-61, 1994.

[W99] D. S. Weld, "Recent advances in AI planning", *AI Magazine*, 20(2): 93-123, 1999.

[Y90] Q. Yang, "Formalizing planning knowledge for hierarchical planning", *Computational Intelligence Journal*, 6(2), pp. 12-24, 1990.

[YBM02] I-Ling Yen, F. B. Bastani, F. Mohamed, H. Ma, and J. Linn, "Application of AI planning techniques to automated code synthesis and testing", In *Proc. 14th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'02)*, November 2002.

[ZW93] S. J. Zeil and C. Wild, "A knowledge base for software test refinement", *Technical Report TR-93-14*, Old Dominion University, Norfolk, VA.