Software Qual J (2007) 15:241–263
DOI 10.1007/s11219-007-9018-3

Rapid goal-oriented automated software testing using MEA-graph planning

Manish Gupta · Jicheng Fu · Farokh B. Bastani · Latifur R. Khan · I.-Ling Yen

Published online: 15 June 2007

Abstract With the rapid growth in the development of sophisticated modern software applications, 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 automated 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. (4) A case study is presented to evaluate the proposed automated testing method and compare the performance of MEA-Graphplan with that of Graphplan. The empirical results show that for software testing, the MEA-Graphplan algorithm can perform computationally more efficiently and effectively than the basic Graph Planning algorithm.

Keywords AI Planning · Planning graph · MEA-Graphplan · Automated software testing

M. Gupta (✉) · J. Fu · F. B. Bastani · L. R. Khan · I.-LingYen
Department of Computer Science, University of Texas at Dallas, Dallas 75083-0688 TX, USA
e-mail: manishg@utdallas.edu

J. Fu
e-mail: jxf024000@utdallas.edu

F. B. Bastani
e-mail: bastani@utdallas.edu

L. R. Khan
e-mail: lkhan@utdallas.edu

I.-LingYen
e-mail: ilyen@utdallas.edu

1 Introduction

During system or integration testing of software systems, it is not only necessary to understand the properties of each of the subsystems 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 (Yen, Bastani, Mohamed, Ma, & Linn, 2002). 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 (Mayrhauser & Hines, 1993; Mayrhauser, Mraz, & Walls, 1994; Mayrhauser, Scheetz, Dahlman, & Howe, 2000). 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 offer great promise because they emphasize goals, i.e., sequences of actions (e.g., plans or test data) are generated specifically to fulfill some purpose. Therefore, the similarities of plans and test cases are that they are both goal-oriented, i.e., both need to conform to the syntactic requirements of the actions/commands and the semantic interactions between actions/commands. The mechanisms of planning are thus ideally suited to test case generation (Howe, Mayrhauser, & Mraz, 1997). Some of the AI planning techniques, including plan-graph planning (Blum & Furst, 1997), plan-space planning (Penberthy & Weld, 1992), HTN planning (Erol, Hendler, & Nau, 1994; Nau, Cao, Lotem, & Muñoz-Avila, 1999), and temporal-logic planning (Bacchus & Ady, 2001; Bacchus & Kabanza, 1996, 2000), can be potential planning techniques for automating the testing process.

Among these planning techniques, Blum and Furst's Graphplan algorithm (1997) 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 (Veloso et al., 1995) and the partial-order planner UCPOP (Penberthy & Weld, 1992). But in the basic Graphplan algorithm, during the graph expansion phase, the planning graph may contain many of the actions that do not contribute to the goal. Thus, the graph expansion algorithm is oblivious of the goal of the planning problem. As a result, during complex system testing, it may experience a higher probability of state-space explosion during the graph expansion phase of the planning. MEA-Graphplan (Kambhampati, Paeker, & Lambrecht, 1997) extends the basic Graphplan algorithm by adapting means-ends analysis (McDermott, 1996) 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 a 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* Σ whose description acts as input parameters to the planning system.
2. The design of a framework for an APS is presented 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 identify certain restrictive assumptions for our APS.
3. We assess 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* Σ for the *List* abstract data type (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.
4. A case study is presented to show how the proposed automated testing method can be applied to a robot simulation system. The performance of MEA-Graphplan is evaluated by comparing it with the basic Graphplan. The empirical evaluation shows that the MEA-Graphplan algorithm can perform computationally more efficiently and effectively than the basic Graph Planning.

The rest of this paper is organized as follows: In Sect. 2, we briefly review various AI planning techniques. In Sect. 3, we formally present the MEA-Graphplan algorithm and explain it in detail. In Sect. 4, we present the conceptual model of a software system from the perspective of testing. In Sect. 5, we propose the automated planning system (APS) framework and provide a comprehensive example for the *List* ADT. Section 6 describes a case study using the proposed automated testing method and presents the performance comparison of MEA-Graphplan and the basic Graphplan. In Sect. 7, we briefly review the related works and in Sect. 8, 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 $< a_1, a_2, ..., a_n >$ 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 (Blum & Furst, 1997) 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 Fig. 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 *zero*th-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 Fig. 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 (Blum & Furst, 1997). Graphplan planning is both *sound* and *complete*.

In plan-space planning (McAllester & Rosenblitt, 1991; Penberthy & Weld, 1992; Weld, 1994), 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*.

HTN planning (Barrett & Weld, 1994; Erol, Hendler, & Nau, 1994; Nau et al., 1999; Yang, 1990) 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 is an appropriate planner that provides a better understanding of the properties of the subsystems. This is because the planning graph constructed during the planning process makes useful constraints, such as interactions and conflicts among subsystems. It also yields an extremely speedy planner that, in many cases, is orders of magnitude faster than total-order and partial-order planners.

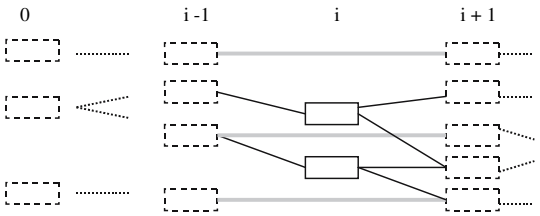

Fig. 1 The Planning graph with action nodes represented by solid-squares, proposition nodes represented by dashed-squares, horizontal gray lines between proposition nodes representing the maintenance actions that encode persistence, and solid lines from proposition nodes to action nodes and vice versa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

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 (Garagnani, 2000; Kambhampati, Paeker, & Lambrecht 1997; McDermott, 1996; Nebel, Dimopoulos, & Koehler, 1997; Weld, 1999) and, as a result, during system testing of complex systems, the graph expansion phase demonstrates a higher probability of state-space explosion.

MEA-Graphplan adapts means-ends analysis (McDermott, 1996) to Graphplan in order to make it goal-oriented. MEA-Graphplan (Kambhampati, Paeker, & Lambrecht, 1997) 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. 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 Fig. 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 Sect. 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.

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). For example, if we define our software system as an ADT or as a process-control

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-levels.
- 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

Fig. 2 MEA-Graphplan algorithm

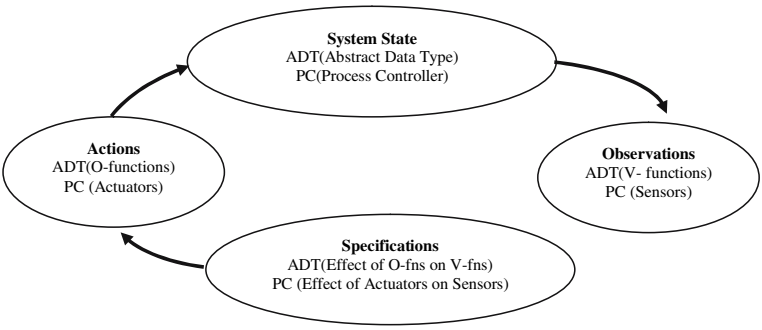

Fig. 3 Conceptual testing-oriented model for software systems

system (PC), then a conceptual testing-oriented model of the system can be represented as shown in Fig. 3.

O−functions (operation-performing functions) and *Actuators* represent methods in each class of the software system that cause transitions in the state space. Also, *V−functions* (value-returning functions) and *Sensors* represent methods in each class of the software system that return 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* Σ:

$\Sigma = \{S, A, T, C, f\}$ where,
$S = \{s_1, s_2, ..., s_m\}$ is a set of states represented using *V*-functions/Sensors,
$A = \{a_1, a_2, ..., a_n\}$ is a set of actions represented using *O*-functions/Actuators,
T = Testing requirements for the software system, e.g., test how each *O*-function affects the *V*-function,
C = Global constraints (corresponding to forbidden regions in the state space), e.g., operators in ADT don't create or destroy the main object.
$f : S \times A \rightarrow 2^S$ 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.

5.1 Automated planning system structure

We propose an APS framework consisting of two components, namely, *Planning Domain Generator* and *AI Planner* as shown in Fig. 4.

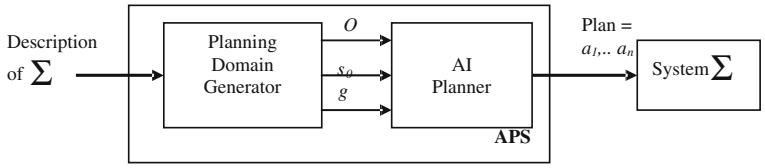

Fig. 4 Automated planning system

5.1.1 Planning Domain Generator

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

We define the software parameters as $\{S', A, T', C, f\}$ where $S' \subseteq S$ and $T' \subseteq T$, and 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.

Definition 1, Initial state of a software system For any planning problem, the initial state s_0 is a set of propositions (literals) called the initial conditions (Blum & Furst, 1997), e.g., the predicates over *V*-functions for ADTs or predicates over Sensors for PCs.

We represent a software system as a collection of finite-state machines where each finite-state model represents the current state of a specific sub-module within it.

Definition 2, Operator set of a software system Each operator defines some domain behavior and consists of an operator name, parameters, preconditions, and effects (add/delete effects). For ADTs, the operator set O is defined using each relevant O-function as an operator in the planning domain. Similarly, for PCs the operator set is defined using relevant Actuators for the system.

While defining an operator's preconditions and effects, the algebraic specification of ADTs are used. Similarly, for PCs, an Actuator's pre-/post-conditions are used.

Definition 3, Goal state of a software system 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 APS can be applied to test software systems with certain restrictive assumptions as discussed in the following definition.

Definition 4, State transition system A state transition system is a tuple $\Sigma = \{S, A, T, C, f\}$ 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., $g \subseteq S$), 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 $< a_1, a_2, ..., a_n >$

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 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 Fig. 5.

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

S = set of states represented using V-functions: { $length(L)$, $ith(L, i)$, $1 \leq i \leq length(L)$},
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 Σ acts as input parameters to the planning system. *APS* component *Planning Domain Generator* maps these input parameters $\{S', A, T', C, f\}$, where $S' \subseteq S, T' \subseteq T$, to the planning parameters $\{O, s_0, g\}$ that are passed as input to the component *AI Planner*. Among these parameters, S', C and T' are mapped to s_0 and g, and A and f are mapped to O. For our example, the component *AI Planner* uses the MEA-Graphplan technique.

V-functions

- *length* : : *List* → *natural*
- *ith* : : *L*: *List x i*: *positive* → *element*
  ![$1 \leq i \leq L.length()$] → *raise Bad Index*

O-functions

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

Fig. 5 Algebraic specification of List ADT

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 Fig. 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 4, 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 1, appends object ''*e*'' at position ''*i*'' in the List, and *shifts* all the element objects at positions $j \geq i$ by *one-place* to the right. The effect of method *append*(*L*, *E*, 3) on the List object is shown in Fig. 6. The method *remove*(*L*, *i*) decreases the length of the List object by 1, 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 Fig. 6.

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 Fig. 7.

Similar to STRIPS-like planning domain (Fikes, & Nilsson, 1971), 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 Fig. 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))*.

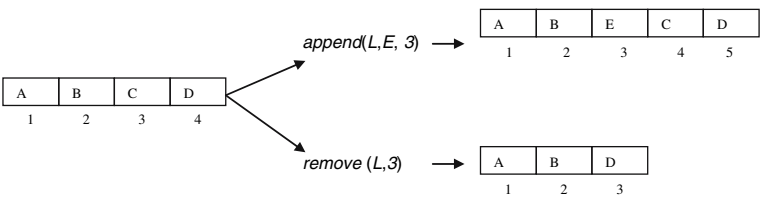

Fig. 6 List Object affected by actions *append*(*L*,*E*,3) and *remove*(*L*,3)

```
(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)))))))))
```

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

5.5 Planning problem

Problem: 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 A1)* ***and*** *(at D4)}*. For the sake of convenience, we use ''*app*'' to denote the operator ''*append*'' and ''*rm*'' to denote ''*remove*'' in the following example.

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

Step 1: Generate the regression-matching graph by regressing *each* goal over actions, till it reaches *initial condition*. Figure 8 shows the regression-matching graph for the initial condition set: *{(length = 3)}*.

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 9 shows the *optimized* forward planning graph constructed.

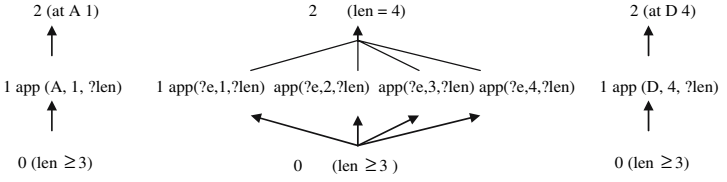

Fig. 8 Regression-match graph with initial conditions at proposition-level 0

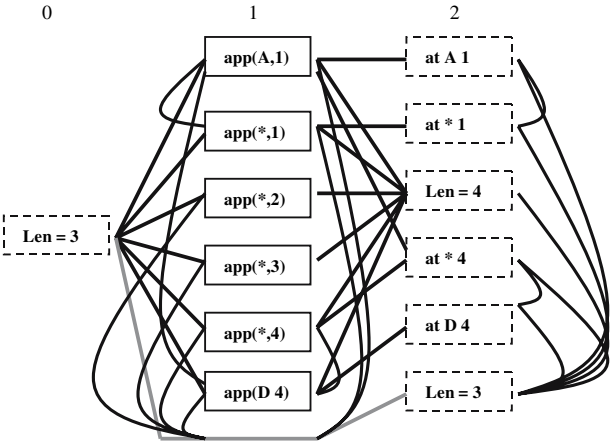

Fig. 9 *Optimized* forward planning graph with action nodes represented by solid-squares, proposition nodes represented by dashed-squares, and horizontal lines between proposition nodes represent the maintenance actions. Thin curved lines between actions (propositions) at a single level denote mutex relations. Some of the no-ops and proposition nodes have not been specified for simplicity

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 A1), (at *1), (at *2), (at *3), (at *4), (len = 4), (at D4), (len = 3)}*. Figure 10 shows the subset of the regression-matching graph for the new initial condition set.

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 Fig. 12 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 A1), (at *1), (at *2), (at *3), (at *4), (at *5), (len = 5), (at D4), (len = 4), (len = 3)}*. Figure 11 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 Fig. 12.

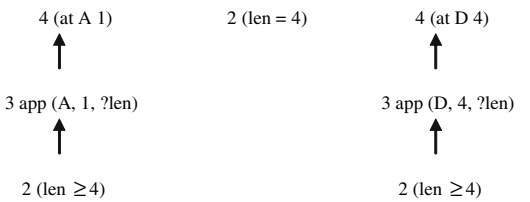

Fig. 10 Regression-match graph subset with initial conditions at proposition-level 2

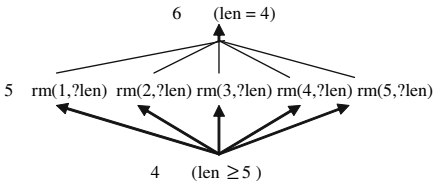

Fig. 11 Regression-match graph subset with initial conditions at proposition-level 4

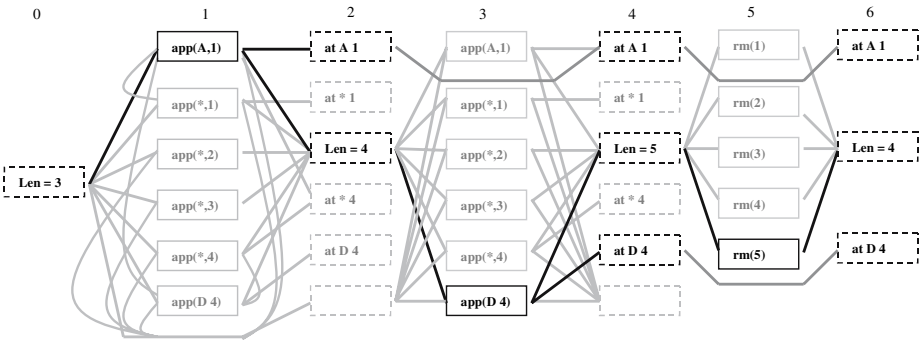

Fig. 12 Optimized forward planning graph with some of the mutex relations and no-ops not shown for simplicity. The valid plan shown as the dark lines in the planning graph

5.6 Summary

We have presented an APS framework and used a comprehensive *List* ADT example to describe how the operators can be developed for the specified planning domain. Our case study, presented in Sect. 6, shows that the MEA-Graphplan algorithm can generate an optimized planning graph and is computationally more efficient and can perform effective solution extraction than the basic Graph Planning algorithm, especially for generating test cases for complex systems such as process control distributed systems.

6 Case study

We have developed a case study to evaluate the proposed goal-oriented automated software testing method.

6.1 System overview

The case study consists of a robot simulation system and the APS. The system operator can control the ‘‘robot‘‘ via a car-like steering wheel and a pedal system. The operator can (a) move the robot at varying speeds along a straight line (b) set the direction of motion to be in the forward or backward direction, and (c) turn the robot around its center. The robot must react to the operator’s actions in real-time.

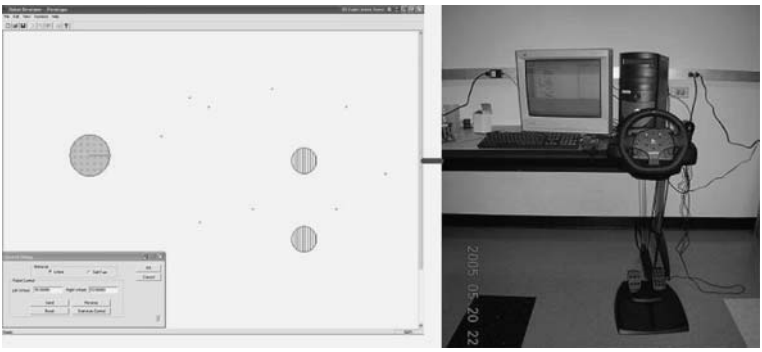

Fig. 13 User interface device and simulator

The robot system simulates a consistent and controllable behavior for the real robot and enables placement of sensors at various points to monitor the state of the system. Figure 13 shows our experimental environment in the case study, consisting of a User Interface Device (UID) and the Simulator interface. The right hand side of Fig. 13 is the joystick, a wheel and pedal, which can control the robot to move along a specified direction at a specified speed. The left hand side is the robot simulator running on the client computer. The simulator has two parts, namely, the status display and a default control interface.

The status display shows the area that the robot can turn or move through. The red circle with a green line within it on the left-top portion of Fig. 13 represents the robot. The direction of the line within the robot represents the direction of the robot's motion (head angle). The robot can be controlled directly by the UID, moving linearly or rotating around its center. Hence, the robot in the simulator behaves exactly in the same way as the real robot. The two black circles on the left-right portion denote the obstacles that the robot should avoid hitting. The red spots in the display interface are waypoints that the robot should move along with.

The interface in the left-bottom portion of Fig. 13 is the default control interface, which is responsible for establishing a socket communication between the robot simulation system and the control site (the computer where the UID/APS resides), and display the speeds of the left and right wheels of the robot. The simulator supports many configuration parameters, including setting of defaults, obstacles, and waypoints. These settings can be combined together to achieve more complex system behaviors.

In order to automatically test the robot simulation system, the APS takes over the control of the robot from the UID (joystick and pedal system). Just like the UID, the APS resides on a different computer than the robot system. As shown in Fig. 14, APS and the robot system communicate with each other through TCP/IP socket links. The robot system sends software parameters (e.g., robot's position, head direction, position of obstacles, testing requirements, etc.) to the control module of APS.

The control module is responsible for communicating with the robot system, dispatching software parameters to the planning domain generator, receiving plans from the AI planner, translating plans into commands for the robot, and monitoring the status of robot.

The Planning Domain Generator then maps these software parameters (robot's position, waypoints information, obstacles information, testing requirements, etc.) to the planning parameters $\{O, s_0, g\}$ (a set of operators, initial state, and goal state) that are passed as inputs to the AI Planner.

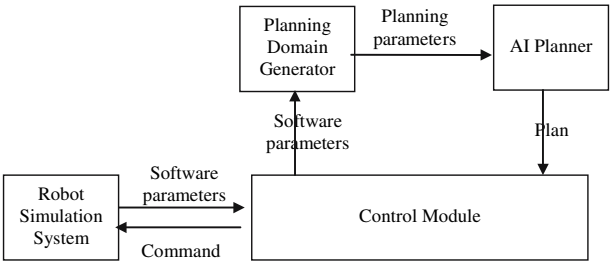

Fig. 14 Test generation process model

We have implemented the MEA-Graphplan as our AI Planner based on an open source graphplan—JPlan (EL-Manzalawy, 2006). After obtaining the planning parameters $\{O, s_0, g\}$, the AI Planner is able to generate a valid plan which is a sequence of instantiated actions $\langle a_1, a_2, ..., a_n \rangle$ from operators in O. The generated plan is then fed to the control module.

6.2 MEA-Graphplan parameters

Essentially, we have three operators in this case study, namely, Turn, Goto, and Flash listed in Table 1.

The robot's location is identified by two parameters: the coordinate of its center and the angle of its head direction. Given a point at coordinate ?*t*, the operator of ''Turn'' functions to turn the robot around its center and adjust its head to point to ?*t*. The predicate of ''align'' is used to check if the robot's head is already directed at the given point. If this is the case, there is no need to execute the ''Turn'' operation; otherwise, the robot will adjust its head direction to an angle calculated by the function of ''calcAngle''.

Table 1 Operators of the robot

Operator	Turn (Robot ?r, Angle ?a, Coordinate ?c, Coordinate ?t)
Constraint	!align(?c, ?t, ?a)
Preconditions	[At(?r, ?c) & heading(?r, ?a)]
Post-conditions	[At(?r, ?c) & heading(?r, calcAngle(?c, ?t))]
Delete-Effect	[heading(?r, ?a)]
Operator	Goto (Robot ?r, Angle ?a, Coordinate ?from, Coordinate ?to)
Constraint	align(?from, ?to, ?a)
Preconditions	[At(?r, ?from) & heading(?r, ?a)]
Post-conditions	[At(?r, ?to)]
Delete-Effect	[At(?r, ?from)]
Operator	Flash (Robot ?r, Coordinate ?c, Obstacle ?o, Coordinate ?t)
Constraint	!Safe(?c, ?t)
Preconditions	[At(?r, ?c) & At(?o,?t)]
Post-conditions	[isFlashing(?r)]

```
Turn ( Rx, <0>, (260,-360), (855,-388) )
Goto ( Rx, <358>, (260,-360), (855,-388) )
Flash ( Rx, (855,-388), o1, (850, -380) )
```

Fig. 15 Generated plan

The operator ''Goto'' is specified as move the robot from its original coordinate ?*from* to a given coordinate ?*to* provided that the robot's head has already pointed to ?*to*. The operator of flash happens when the robot hits an obstacle. The predicate ''Safe'' is used to check if the robot's current location is far enough from the given obstacle. If the robot is within the unsafe region, the robot would start to flash, which serves as a warning sign. Actually, the unsafe region around the obstacles is what we called ''forbidden regions''. By specifying the goal states that are close to the forbidden region, our planning technique can automate the testing process. Therefore, with the help of APS, we just need to decide what to test, and let the planning system determine how to do it.

For example, initially, the robot is located at Coordinate (260, −360) with head angle of 0. The two obstacles are located at (850, −380) and (850, −580). The goal is to move the robot to a location with Coordinate (855, −388) within the forbidden region and the robot is flashing. With these initial and goal states, our AI planner produces a plan shown in Fig. 15, which starts with turning the head of the robot to direct to the Coordinate (855, −388), then moves from (260, −360) to (855, −388), and, finally, because it is now in the forbidden region, the robot starts to flash.

Consider that the test engineer may want the robot to automatically follow a route to reach the destination. The robot simulation system allows the tester to specify a sequence of waypoints for the robot to move along with. If some accident occurs (e.g., hitting an obstacle), the joystick will be used to control the robot manually to leave the forbidden region.

We therefore extend our definitions of the operators in Table 1 to accommodate the specification of waypoints. First, the Coordinate (x, y) is extended to three dimensions, i.e., (x, y, i), where x and y have the same meaning as the original coordinate, and i indicates the index of the specified waypoint among all the waypoints. The definitions of operators for waypoints are listed in Table 2.

Table 2 Operators supporting waypoints

Operator	Turn (Robot ?r, Angle ?a, Waypoint ?c, Waypoint ?t)
Constraint	!align(?c, ?t, ?a) & sequent(?c,?t)
Preconditions	[At(?r, ?c) & heading(?r, ?a)]
Post-conditions	[At(?r, ?c) & heading(?r, calcAngle(?c, ?t))]
Delete-Effect	[heading(?r, ?a)]
Operator	Goto (Robot ?r, Angle ?a, Waypoint ?from, Waypoint ?to)
Constraint	align(?from, ?to, ?a) & sequent(?from,?to)
Preconditions	[At(?r, ?from) & heading(?r, ?a)]
Post-conditions	[At(?r, ?to)]
Delete-Effect	[At(?r, ?from)]

```
Objects:
Robot (Rx);
Angle (<20>);
Waypoint ((260,-360,0));
Waypoint ((426,-299,1));
Waypoint ((575,-213,2));
Waypoint ((759,-325,3));
Waypoint ((682,-446,4));
Waypoint ((472,-495,5));
Init: At (Rx,(260,-360,0)) & heading(Rx,<20>)
Goal: At (Rx,(472,-495,5))
```

Fig. 16 Example waypoint problem

```
Goto ( Rx, <20>, (260,-360,0), (426,-299,1) )
Turn ( Rx, <20>, (426,-299,1), (575,-213,2) )
Goto ( Rx, <29>, (426,-299,1), (575,-213,2) )
Turn ( Rx, <29>, (575,-213,2), (759,-325,3) )
Goto ( Rx, <329>, (575,-213,2), (759,-325,3) )
Turn ( Rx, <329>, (759,-325,3), (682,-446,4) )
Goto ( Rx, <237>, (759,-325,3), (682,-446,4) )
Turn ( Rx, <237>, (682,-446,4), (472,-495,5) )
Goto ( Rx, <193>, (682,-446,4), (472,-495,5) )
```

Fig. 17 Generated plan for the waypoint problem

When the number of waypoints is large, the state-space of graph expansion can grow exponentially. Therefore, we use the predicate ''sequent'' to limit the search space to each pair of successive waypoints. In other words, the robot can only turn to or go to the succeeding waypoint.

Consider the following example to illustrate the waypoint testing process. Assume that the tester has specified six waypoints. Parts of the planning parameters, which are automatically generated by APS, are listed in Fig. 16.

The generated plan from our MEA-Graphplan AI Planner is: (Fig. 17)

In the current implementation, the testing request is initiated from the robot simulation system. For example, after the test engineer completes setting the waypoints, there is a menu in the interface for him or her to send the testing request along with other software parameters (e.g., the waypoints settings, the robot's location, head angle etc) to the control module of the APS.

Another issue worthy of notice is that each step of the test case execution should be monitored and verified. An incorrect state can lead to an unexpected result which makes subsequent steps useless. Therefore, the APS should determine when to terminate the testing process as soon as an exception is detected. In the current implementation, an exception is detected when the socket connection is interrupted or the robot system is not responsive for a certain period of time. The APS can keep track of the last successful communication message from the robot system, the last successfully executed command, and the command under execution. All these information are good clues for the system developers to figure out what is going on with the robot simulation system under testing.

6.3 Empirical evaluation

We have implemented the MEA-Graphplan and compared its performance with the Graphplan on our Robot domain. The performance of Graphplan is thoroughly evaluated in

Blum and Furst (1997), which shows that, in some cases, it is orders of magnitude faster than other planning techniques. So the good performance of MEA-Graphplan in the following two sets of experimental comparisons with Graphplan further justifies its applicability as the AI Planner for our automated testing method.

The first set of experiments tries to reveal the effect of irrelevant literals in the initial state. Some prior works (EL-Manzalawy, 2006; Parker, 1999) that aim to improve the original Graphplan have demonstrated that the algorithms applying the regression-matching graph are not affected by the irrelevant initial literals. However, our empirical evaluation shows that this is only true if no new objects irrelevant to the goal are introduced to the planning problem.

Consider the waypoint problem in Fig. 16 as an example to show the effects of irrelevant initial literals. The initial state of the waypoint problem is that Robot *Rx* is located at Waypoint (260, −360, 0) with head angle of 20. The objects in this problem include Robot *Rx*, angle ''20'', and six waypoints.

We introduce a new object, Robot *Ra*, to the planning problem. We add to the initial states a literal At (*Ra*, (260, −360, 0)) which is irrelevant to the goal state. From the curve of MEA-Graphplan in Fig. 18 we can see that the corresponding time to produce a valid plan increases a little as the number of irrelevant initials increases from 0 to 1.

The reason is that the instantiated irrelevant operators contribute to the performance overhead. For example, the operator Turn(Robot ?r, Angle ?a, Waypoint ?c, Waypoint ?t) can be instantiated to Turn(Robot *Ra*, ⟨20 ⟩, (260, −360, 0), (426, −299, 1)), Turn(Robot *Ra*, ⟨20 ⟩, (426, −299, 1), (575, −213, 2)) ... During the construction of the regression-matching graph in MEA-Graphplan, each of the instantiated actions will be chosen to verify its applicability to the current proposition level. Because Robot *Ra* is irrelevant to the goal At(*Rx*, (472, −495, 5)), these irrelevant instantiated operators would never be added to the graph. Therefore, the MEA-Graphplan will not suffer from the state-space explosion as Graphplan does. However, the process of verifying the applicability of the

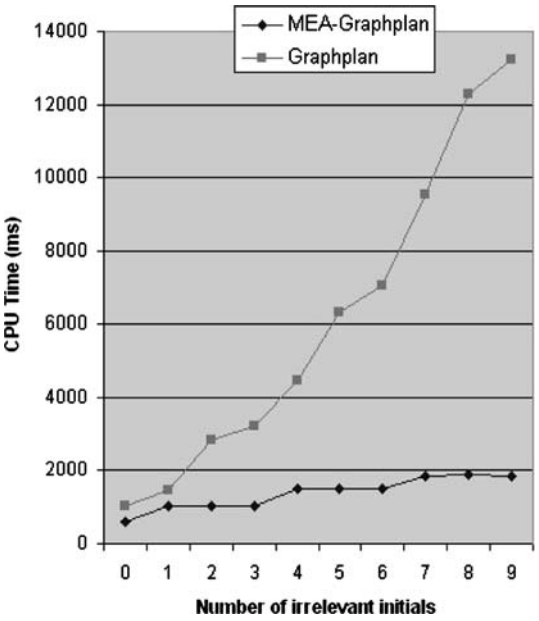

Fig. 18 Effect of irrelevant literals in initial state

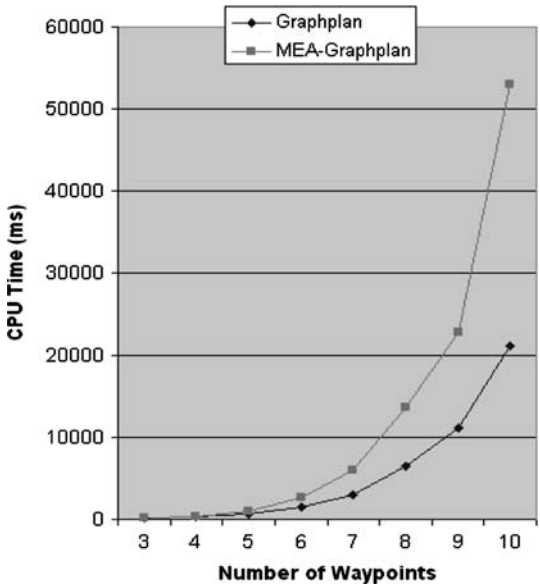

Fig. 19 Effect of number of waypoints

irrelevant instantiated operators does incur some overhead. That is why the total planning execution time increases slightly.

To confirm our analysis, we add another irrelevant literal to the initial state, heading $(Ra,\langle 20\ \rangle)$, with no new objects introduced. This time, the planning execution time remains almost the same as the proceeding test. Then, we add the third irrelevant literal, heading $(Ra,\langle 193\ \rangle)$, without new objects added too. The planning execution time once again remains almost the same. We continue the experiment and repeat the same process by adding new objects, Robot *Rb* and *Rc*, shown in Fig. 18 corresponding to the number of irrelevant initials of 4 and 7 respectively. Then we observed that the same pattern repeats. Meanwhile, the time of Graphplan to produce a valid plan grows substantially as the number of irrelevant initial literals increases.

In the second set of experiments, we compared the performance of the two planners in solving the waypoint problem by setting a set of varying number of waypoints. As shown in Fig. 19, when the number of waypoints is less than 4, the two planners perform almost the same. The graphplan performs even better than MEA-Graphplan because the cost of constructing the regression-matching graph counteracts the benefit of MEA-Graphplan for simple cases. However, as the number of waypoints increases, the performance of Graphplan decreases rapidly relative to that of MEA-Graphplan. As a result, the curve of its execution time grows much faster than that of MEA-Graphplan.

7 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 Fickas and Anderson (1988) and Anderson (1993), 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 Fickas and Helm (1992), 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 Huff and Lesser (1998) and Huff (1992), 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 Rist (1992), 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 Deason, Brown, Chang, and Cross (1991), the authors used rule based test generation method that encodes white box criteria and information about control and data flow of the code. In Chilenski and Newcomb (1994), the authors used a resolution-refutation theorem prover to determine structural test coverage and coverage feasibility. In Zeil and Wild (1993, the authors used a knowledge base of entities, their relationships, and their refinements for refinement of the test case description. Anderson, Mayrhauser, and Mraz (1995) 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 Howe, Mayrhauser and Mraz (1997), Scheetz, Mayrhauser, France, Dahlman, and Howe (1999) and Mraz, Howe, Mayrhauser, and Li (1995), the authors use the 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 Veloso and (Blythe, 1994).

In Memon, Pollack, and Soffa (2001), the authors use Interference Progression Planner (IPP) and HTN planning for test case generation. IPP (Anderson, Smith, & Weld, 1998; Koehler, Nebel, & Hoffmann, 1997), 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 (Veloso et al., 1995) and the partial-order planner UCPOP (Penberthy, & Weld, 1992). HTN planning, also referred to as Hierarchical planning, is 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 Gupta, Bastani, Khan, and Yen (2004), 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 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, the computationally efficient and effective MEA-Graph planning greatly alleviates the problem of state-space explosion during the graph expansion phase of the planning process. In Gupta et al. (2004), we also propose an APS for applying any of the available AI planning techniques for software testing of an ADT.

8 Conclusion and future work

During automated software testing, AI planning techniques Graphplan and its descendents MEA-Graph planning 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, our case study indicates that MEA-Graph planning is potentially computationally more efficient and effective than basic Graph Planning, especially for complex system testing.

We have presented a comprehensive *List* ADT example to show how MEA-Graphplan can work for the specified planning domain. However, the proposed APS is not limited to ADTs. Essentially, it can be applied to any state transition system, $\Sigma = \{S, A, T, C, f\}$, with a finite set of states S, controllable actions A, and deterministic state-transition functions f.

To apply our proposed APS, the state transition system Σ and the mapping from software parameters to planning parameters should be defined, and some domain specific predicates and functions (e.g., the predicate ''Align'' and function ''calcAngle'' in our case study) need to be incorporated into the APS. We have formulated a framework for automated software testing, which can be applied to a specific domain by incorporating domain-specific information (e.g., predicates and functions). We have also evaluated the performance of the MEA-Graph planning technique in comparison with Graphplan planning technique using two comprehensive sets of examples in the robot domain, and the empirical results show that the performance of MEA-Graphplan is relatively better.

One potential future research direction is to explore the application of some learning techniques (Borrajo & Veloso, 1996; Minton et al., 1989; Veloso, 1994) in AI planning for automated software testing. Learning in AI planning can basically be categorized into learning in total-order planner (Avila et al., 2001; Veloso, 1994; Veloso et al., 1995) and learning in partial-order planner (Estlin & Mooney, 1996; Muñoz-Avila, 1998; Muñoz-Avila & Weberskirch, 1997). During unit testing of subsystems, we can train the AI planner to generate plans for individual subsystems. This training can speed up the planning process during system or integration testing and lead to more efficient and faster planning for automated software testing.

References

Avila, H. M., Aha, D. W., Nau, D. S., Weber, R., Breslow, L., & Yaman, F. (2001). SiN: Integrating case-based reasoning with task decomposition. *IJCAI*-2001, Seattle, August.

Anderson, J. S. (1993). Automating requirements engineering using Artificial Intelligence Techniques. Ph.D. thesis, Dept. of Computer and Information Science, University of Oregon.

Anderson, C., Mayrhauser, A., & Mraz, R. (1995). On the use of Neural Networks to guide Software Testing Activities. In *Proc. of International Test Conference*, Washington, DC.

Anderson, C., Smith, D. E., & Weld, D. (1998). Conditional effects in graphplan. In *Proc. of 4th Intl. Conf. on AI Planning Systems*, June.

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

Blum, A., & Furst, M. (1997) Fast planning through planning graph analysis. *Artificial Intelligence, 90*, 281–300.

Bacchus, F., & Kabanza, F. (1996). Using temporal logic to control search in a forward chaining planner. In: M. Ghallab & A. Milani (Eds.), *New directions in planning* (pp. 141–153). IOS Press.

Bacchus, F., & Kabanza, F. (2000). Using temporal logic to express search control knowledge for planning. *Artificial Intelligence, 116*, 123–191.

Borrajo, D., & Veloso, M. M. (1996). Lazy incremental learning of control knowledge for efficiently obtaining quality plans. *AI Review Journal. Special Issue on Lazy Learning, 10*, 1–34.

Barrett, A., & Weld, D. (1994). Task-decomposition via plan parsing. In *Proc. of AAAI*–94, Seattle, WA, July.

Chilenski, J. J., & Newcomb, P. H. (1994). Formal specification Tools for Test Coverage Analysis. In *Proc. of Ninth Knowledge-Based Software Engineering Conference* (pp. 59–68). Monterey, CA.

Deason, W., Brown, D., Chang, K. H., & Cross, J. (1991). Rule-based software test data generator. *IEEE Transactions on Knowledge and Data Engineering, 3*(1), 108–117.

Erol, K., Hendler, J., & Nau, D. S. (1994). 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.

Estlin, T. A., & Mooney, R. J. (1996). Hybrid learning of search control for partial-order planning. *New Directions in AI Planning* (pp. 129–140). IOS Press.

Fickas, S., & Anderson, J. (1988). 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.

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

Fikes, R., & Nilsson, N. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence, 2*(3/4), 189–208.

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

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

Huff, K. (1992). Software adaptation. In *Working Notes of AAAI-92 Spring Symposium on Computational Considerations in Supporting Incremental Modification and Reuse* (pp. 63–66). Stanford University.

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

Howe, A., Mayrhauser, A., & Mraz, R. (1997). Test case generation as an AI planning problem. *Automated Software Engineering, 4*(1), 77–106.

Koehler, J., Nebel, B., Hoffmann, J., & Dimopoulos, Y. (1997). Extending planning graphs to an ADL subset. In *Proc. 4th European Conference on Planning* (pp. 273–285). September.

Kambhampati, R., Paeker, E., & Lambrecht, E. (1997). Understanding and extending graphplan. In *Proc. 4th European Conference on Planning*, September.

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

Muñoz-Avila, H. (1998). *Integrating twofold case retrieval and complete decision replay in CAPlan/CbC*. PhD Thesis, University of Kaiserslautern.

EL-Manzalawy, Y. (2006). Efficient planning with initial irrelevant facts. http://www.cs.iastate.edu/~yasser/cs572pro.html (Feb 15, 2006)

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

Mayrhauser, A., & Hines, S. C. (1993). Automated testing support for a robot tape library. In *Proc. of the Fourth International Software Reliability Engineering Conference* (pp. 6–14). November.

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

Mayrhauser, A., Mraz, R. T., & Walls, J. (1994). Domain based regressing testing. In *Proc. of the International conference on Software Maintenance* (pp. 26–34). Sept.

Memon, A. M., Pollack, M. E., & Soffa, M. L. (2001). Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering, 27*(2) February.

McAllester, D., & Rosenblitt, D. (1991). Systematic nonlinear planning. In *Proc. 9th National Conference on Artificial Intelligence*.

Mayrhauser, A., Scheetz, M., Dahlman, E., & Howe, A. E. (2000). Planner based error recovery testing. In *Proc. 11th International Symposium on Software Reliability Engineering* (pp. 186–195). Oct.

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

Nau, D., Cao, Y., Lotem, A., & Muñoz-Avila, H. (1999). SHOP: Simple hierarchical ordered planner. *IJCAI, 99*, 968–973.

Nebel, B., Dimopoulos, Y., & Koehler, J. (1997). Ignoring irrelevant facts and operators in plan generation. In *Proc. 4th European Conference on Planning*, Sept

Parker, E. (1999). Making Graphplan Goal-Directed. In *Proceedings of the 5th European Conference on Planning: Recent Advances in AI Planning*, pp. 333–346.

Penberthy, J. S., & Weld, D. (1992). UCPOP: A sound, complete, partial order planner for ADL. In *Proc. 3rd Intl. Conf. Principles of Knowledge Representation and Reasoning*(pp. 103–114). Oct.

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

Scheetz, M., Mayrhauser, A., France, R., Dahlman, E., & Howe, A. (1999). 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.

Veloso, M. M. (1994). 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: AAAI Press.

Veloso, M. M., & Blythe, J. (1994). Linkability: Examining causal link commitments in partial-order planning. In *Proc. of the Second International Conference on AI Planning Systems* (pp. 170–175). June

Veloso, M. M., Carbonell, J., Perez, M. A., Borrajo, D., Fink, E., & Blythe, J. (1995). Integrating planning and learning: The prodigy architecture. *Journal of Experimental and Theoretical Artificial Intelligence, 7(1)*, 81–120.

Weld, D. S. (1994). An Introduction to least-commitment planning. *AI Magazine, 15*(4), 27–61.

Weld, D. S. (1999). Recent advances in AI planning. *AI Magazine, 20*(2), 93–123.

Yang, Q. (1990). Formalizing planning knowledge for hierarchical planning. *Computational Intelligence Journal, 6*(2), 12–24.

Yen, I-L., Bastani, F. B., Mohamed, F., Ma, H., & Linn, J. (2002). 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.

Zeil, S. J., & Wild, C. (1993). A knowledge base for software test refinement. *Technical Report TR-93-14*, Old Dominion University, Norfolk, VA.

Author Biographies

Manish Gupta received his B.E. degree in Electrical Engineering from Govindram Sakseria Institute of Technology & Sciences, India in 1997 and his M.S. degree in Computer Science from University of Texas at Dallas in 2002. His current research interest includes AI-based software synthesis and testing. His other research interests include mobile computing, aspect-oriented programming and model checking. He is currently working as Software Designer with Rudolph Technologies and has been involved in designing and developing, automated macro defect inspection tools used by semiconductor device manufactures worldwide.

Jicheng Fu received his BS degree in Computer Science in 1996 from Harbin Institute of Technology, P.R. China. He received his MS degree in Computer Science in 1998 from Harbin Institute of Technology, P.R. China. He is now a PhD student in Computer Science at the University of Texas at Dallas. His primary research interest is on Automated Program Synthesis.

Dr. Farokh B. Bastani received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Bombay, and the M.S. and Ph.D. degrees in computer science from the University of California, Berkeley. He is currently a professor of computer science at the University of Texas at Dallas. His research interests are in the areas of relational programs, high-assurance hardware/software systems engineering, hardware/software reliability assessment, self-stabilizing systems, inherent fault tolerance, and high-performance modular parallel programs.

Dr. Bastani has been on the program committees of numerous conferences and workshops and the editorial board of the IEEE Transactions on Software Engineering and the High Integrity Systems Journal. He is currently an emeritus Editor-in-Chief of the IEEE Transactions on Knowledge and Data Engineering, on the editorial boards of the International Journal of Artificial Intelligence Tools and the Journal of Knowledge and Information Systems (KAIS), and on the Steering Committees of the IEEE International Conference on Tools with Artificial Intelligence and the IEEE Symposium on Service-Oriented Systems Engineering. He is a member of ACM and the IEEE.

Dr. Latifur R. Khan is currently an Associate Professor in the Computer Science department at the University of Texas at Dallas (UTD), where he has taught and conducted research since September 2000. His research work is currently supported by grants from the Air Force Office of Scientific Research (AFOSR), National Science Foundation (NSF), the Nokia Research Center, Alcatel, and the SUN Academic Equipment Grant program. Dr. Khan is one of the principal investigators at the CyberSecurity and Emergency Preparedness Institute at UTD, where he is involved with finding solutions to deal with the rapidly growing Homeland Security problems in cyber crime, information assurance, and emergency preparedness. In addition, Dr. Khan is the director of the state-of-the-art DBL@UTD, UTD Data Mining/Database Laboratory, which is the primary center of research related to data mining and image/video annotation at University of Texas-Dallas. Dr. Khan's research areas cover data mining, multimedia information management, semantic web and database systems with the primary focus on first three research disciplines. He has served as a committee member in numerous prestigious conferences, symposiums and workshops including the ACM SIGKDD Conference on Knowledge Discovery and Data Mining. Dr. Khan currently serves on the editorial board of North Holland's Computer Standards and Interface Journal, Elseiver Publishing. Dr. Khan has published over 80 papers in prestigious journals and conferences.

Dr. I-Ling Yen received her BS degree from Tsing-Hua University, Taiwan, and her MS and PhD degrees in Computer Science from the University of Houston. She is currently an Associate Professor of Computer Science at the University of Texas at Dallas.

Dr. Yen's research interests include fault-tolerant computing, security systems and algorithms, distributed systems, Internet technologies, E-commerce, and self-stabilizing systems. She had published over 100 technical papers in these research areas and received many research awards from NSF, DOD, NASA, and several industry companies. She has served as Program Committee member for many conferences and Program Chair/Co-Chair for the IEEE Symposium on Application-Specific Software and System Engineering & Technology, IEEE High Assurance Systems Engineering Symposium, IEEE International Computer Software and Applications Conference, and IEEE International Symposium on Autonomous Decentralized Systems. She is a member of the IEEE.