Automated Discovery of Loop Invariants for High-Assurance Programs Synthesized Using AI Planning Techniques

Jicheng Fu, Farokh B. Bastani, I-Ling Yen
Department of Computer Science
University of Texas at Dallas
{*jxf*024000@*utdallas.edu*, *bastani@utdallas.edu*, *ilyen@utdallas.edu*}

Abstract

The discovery of loop invariants is a great challenge for the independent verification of automatically synthesized programs. This verification is needed to achieve high confidence in the correctness of the synthesized code, i.e., assurance that no latent defects in the synthesizer itself could have led to the synthesis of an incorrect program. To address this problem, we present an automated loop invariant discovery approach for programs synthesized using a combination of AI planning and component-based software development techniques. Specifically, a plan (denoting the synthesized code) is generated by an enhanced Graphplan planner first. The loop invariants can be automatically discovered based on the same planning graph used to synthesize the code. The correctness can be independently verified via standard loop invariant proof steps, including initialization, maintenance, and termination. The proposed approach not only has a rigorous theoretical basis, but is also guaranteed to produce accurate invariants by removing spurious invariants that are independent of the concerned loop. In combination with other loop invariant detection techniques, the proposed approach can produce loop invariants for complex programs and, thus, greatly facilitate high-confidence automated verification of synthesized systems.

1. Introduction

Loop invariants play an essential role in initial software development as well as subsequent software evolution and maintenance. Being fully aware of loop invariants, programmers are less likely to violate the properties that ensure the correct behaviors of the software under construction. This can improve the quality of manually composed programs. On the other hand, such invariants are equally critical for verifying programs that are automatically generated by some program synthesis tools. For example, NASA did not attempt to prove the correctness of the program generator for Kalman Filters because of the complexity, but came up with a technique to verify the generated code instead [8]. Loop invariants are used to verify the correctness of the generated Kalman Filters. Although there are program synthesis methods that are designed to be "correct-by-construction", such as AMPHION [21], the program generator itself could have implementation bugs in it. Therefore, using a different technique to independently cross-verify the generated programs is needed to enhance the confidence in the reliability of the program.

AI planning is attractive for automated software engineering because of its emphasis on goals and the similarity of plans to programs. There are research works [17][22] demonstrating that the formalization of component-based software development (CBSD) shows great similarities to the problem of AI planning. It is very promising to combine these two techniques to achieve automated program synthesis. This requires AI planners to be powerful in generating glue code. However, the majority of existing AI planners can only generate sequential plans, in which no conditional and loop constructs are possible. For the few AI planners that are capable of generating conditional and loop constructs, they are either not efficient or not sufficiently scalable. Hence, we design and implement a fast iterative planner, FIP [7], which extends classical Graphplan [2] and can achieve high scalability and efficient planning. It can generate procedure-like generic reusable plans, called procedural plans. The techniques used in FIP can be applied to other Graphplan variants as well. In this sense, the big Graphplan family can be enhanced with the capability of iterative planning.

Figure 1 illustrates the architecture of the hybrid program synthesis system, in which the AI planning subsystem is at the core. The generated plan can be seen as the synthesized code that chooses and organizes the underlying components to achieve a certain goal. In this paper, we present a novel approach for discovering

1530-2059/08 $25.00 © 2008 IEEE
DOI 10.1109/HASE.2008.36

loop invariants to facilitate the verification of the synthesized program.

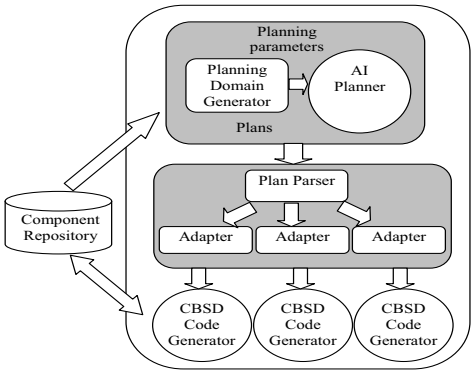

Figure 1: Architecture of Planning-Based Code Synthesis System

This approach is based on the analysis of the planning graph, in which a plan is generated. It exploits Graphplan's intrinsic features, e.g., level-off, layered plan, etc., to discover loop invariants. Specifically, right before the loop starts, a set of propositions is identified as the possible invariant. This set is further refined by checking its validity before each iteration and at termination. This typically fulfills all the standard invariant proof steps, including initialization, maintenance, and termination. Hence, the proposed approach is theoretically sound.

The proposed method discovers invariants for the code that glues the underlying components together. It assumes that the underlying components are correct. Other dynamic invariant detection techniques, e.g., Daikon [4], can be used to verify the correctness of the underlying components. Therefore, this typically forms an invariant discovery hierarchy with the proposed approach leveraging the capabilities of other invariant detection techniques. This can greatly improve the overall system's reliability.

In addition, AI planning is extensively used in service-oriented composition and workflow generation [18]. The proposed approach can be seamlessly applied to this area also. Invariants can be automatically discovered and used to ensure the correctness of the composite service or synthesized workflow.

The rest of this paper is organized as follows: In Section 2, some background knowledge is introduced and the Graphplan properties that facilitate loop generation and loop invariant discovery are presented. In Section 3, the algorithm that supports the synthesis of conditional and loop constructs is presented. Two propositions are given for identifying the loop boundary. In Section 4, we present the AI planning-based loop invariant discovery approach. In Section 5, the related works are reviewed and in Section 6, we conclude the paper and identify some future research directions.

2. Background

This section briefly introduces the background knowledge regarding how AI planning and CBSD can be integrated together so that the requisite glue code can be synthesized and how loop invariants are identified as a by-product of the program synthesis process.

A software component can be modeled as a planning operator because it is an independent unit that provides a predefined service. Its behavior can be captured by constraints that must be satisfied before using the component (preconditions) and the conditions that will be established by the execution of the component (post-conditions). As shown in Figure 1, a component repository is available to the system so that the planning domain generator can navigate through it and generate planning parameters for the AI planner. Thus, the AI planner can work over the planning operators that are converted from components and select and organize the components based on the planning parameters and eventually form a plan that can be used to derive the glue code. Then, the plan parser parses the generated plan and passes the uniformly formatted one to adapters that work as bridges between the AI planning system and specific CBSD systems. Finally, the CBSD code generator will help generate the final codes.

Loop invariants can be generated along with the planning process. We have implemented an enhanced Graphplan planner, FIP, that can generate program-like plans, called procedural plans. Since the enhanced planner involves Graphplan, we first briefly introduce the basic concepts of Graphplan.

2.1. Graphplan

Graphplan alternates between graph expansion and solution extraction phases. During the graph expansion phase, the planning graph is extended in the forward direction 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 to identify a valid plan. The generated planning graph is arranged in levels alternating between proposition and action levels. "No-op" actions propagate the propositions from the current proposition level to the next proposition level.

Mutually exclusive (mutex) relations among actions/propositions should be identified and propagated. No two actions at the same level in a valid plan can be mutex. If no appropriate 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 failure. Graphplan is both sound and complete [2].

2.2. Notational Conventions

To better illustrate the idea of the planning method, we use STRIPS planning scheme to formulate planning problems. But our method is not limited to STRIPS domains. It can be applied to PDDL [16] planning domains as well.

We follow the convention in [11] that all operator schemata are supposed to be grounded, i.e., actions. A STRIPS action α is defined in a triple, $\alpha = \langle pre(\alpha), add(\alpha), del(\alpha)\rangle$, where $pre(\alpha)$ are the preconditions of α; $add(\alpha)$ are the add effects of α; and $del(\alpha)$ are the delete effects of α.

In addition, we define A_i as the i-th action level and P_i as the i-th proposition level. The set of mutex pairs in A_i is defined as $\mu A_i = \{(\alpha, \beta) \mid \alpha, \beta \in A_i$ and α and β are mutex$\}$. Similarly, $\mu P_i = \{(p, q) \mid p, q \in P_i$ and p and q are mutex$\}$. Then, a planning graph G that is expanded up to level i is defined as follows [15]:

$G = \langle P_0, A_1, \mu A_1, P_1, \mu P_1, .., A_i, \mu A_i, P_i, \mu P_i\rangle$

2.3. Planning Graph Properties for Loop Generation and Invariants Detection

Planning graph structures exhibit many favorable properties that provide adequate information for reachability analysis, loop generation, and invariants detection.

2.3.1. Level-Off. In classical Graphplan, "level-off" is a feature that guarantees the termination of the planning process when no plans exist. As shown in Figure 2, level-off takes place when the proposition level P_i is identical to P_{i+1}. If we continue to expand the planning graph, then the next action level A_{i+2} will be identical to A_{i+1}. Action level A_{i+2} will generate another proposition level which is identical to P_{i+1} and the same pattern repeats forever. Hence, "level-off" represents an endless loop in which all applicable actions have been used, but cannot generate a desirable effect exiting the loop. This feature can be exploited to facilitate the generation of loop constructs, which is discussed in Section 3.

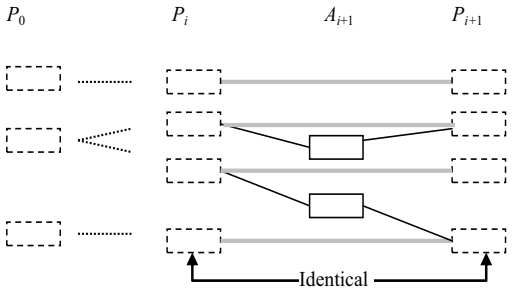

Figure 2: Level-off

2.3.2. No-op. No-op actions are designed to propagate propositions from the current proposition level to the next proposition level. One no-op action just propagates one proposition at each time. No-ops provide the planning graph with the property that if proposition $p \in P_i$, then $p \in P_j$ for all $j \geq i$. As shown in Figure 2, the thick grey lines represent no-ops.

This property gives us a hint that loop invariants can be discovered with the help of no-ops. Specifically, if we know that a loop L starts in proposition level P_i and ends in proposition level P_j ($j > i$), we can obtain a set of propositions $\wp \in P_i$, such that the propositions in $\wp$ are not mutex with the preconditions of loop L. As no-ops will propagate $\wp$ to P_j, we can check whether $\wp$ is not mutex with the outcomes of loop L in P_j as well. If it is the case, then the loop invariant properties of initialization and termination have been fulfilled. What is left is to verify the property of maintenance. Section 3 discusses in detail how to identify a loop's starting and ending points and Section 4 discusses how to verify the properties of initialization, maintenance, and termination.

2.4. Nondeterministic Actions

To motivate nondeterminism in planning, and to explain the concepts in subsequent sections, we use the joystick control problem [5] as our running example. This is a tele-control robot simulation system where the system operator manipulates a joystick to remotely control a robotic system. Assume that there is a button on the joystick that can be used by the operator to send the termination command to halt the robot. The planning problem is to read all the normal control commands sent by the operator through the joystick until the termination command is issued, and then the joystick is closed. The action "ReadJoystick" is nondeterministic because the received command may be a control command or a termination command. The

time at which the operator will send out the termination command is nondeterministic. Now, let us formally define a nondeterministic planning domain.

Definition 2.1 A nondeterministic planning domain is a 4-tuple $\Sigma = (P, S, A, \gamma)$, where:

- P is a finite set of propositions;
- $S \subseteq 2^P$ is a finite set of states in the system;
- A is a finite set of actions that are fully instantiated operators;
- $\gamma: S \times A \rightarrow 2^S$ is the state-transition function.

According to this definition, a nondeterministic action can have multiple effects, each of which is a state in S. One of the effects is called the *intended effect* and represents the desired effect to be generated, while others are called *failed effects* [12], representing action failures or undesired external events. For example, if the goal is to close the joystick, then the intended effect for "ReadJoystick" is "the termination command is issued" and the failed effect is "read a normal control command".

In order to formulate the nondeterministic action that has multiple effects with STRIPS, we apply a method similar to [9], i.e., a nondeterministic action with multiple effects is decomposed into multiple actions. Formally, suppose *act* is an action having multiple possible effects $E = \{e_1, e_2, \ldots, e_n\}$. Then, $\forall e$: $e \in E$, create an action α such that,

- $pre(\alpha) = pre(act)$ and,
- $add(\alpha) = e$.

In order to differentiate these decomposed actions from the original actions that have a single effect, we define the set of decomposed actions as ***D-action***s and other actions as ***G-action***s. The difference is that D-actions are decomposed from some actions with multiple effects, while G-actions are the originally existing standard classical actions.

Definition 2.2 Given an action *act* with multiple effects, the decomposition function δ is defined as $\delta(act) = \{\alpha \mid \alpha$ is the action obtained from the decomposition of action *act*$\}$.

Definition 2.3 Let *act* be an action with multiple effects. If there is a D-action $\alpha \in \delta(act)$ that creates the intended effect at a certain time step, then α is called an *S*-action. All D-actions that generate failed effects are called F-actions.

It should be noted that S-actions and F-actions are NOT static. At different time steps, previous S-actions could become F-actions and vice versa. In summary, for a planning problem, three kinds of planning actions are possible:

- G-action: This is the set of standard Graphplan actions.
- F-action: This includes D-actions that generate failed effects at a certain time step.
- S-action: This includes special D-actions that generate the intended effects at a certain time step.

3. Loop Identification

In a nondeterministic domain, each failed effect needs some "remedial actions" to fix the effect. In [7], a novel algorithm that extends classical Graphplan is proposed to support both conditional as well as loop constructs. An enhanced Graphplan planner, FIP, has been developed to implement the algorithm.

FIP is a two-phase planning algorithm with two novel concepts. First, it makes use of the level-off property for loop construction. It relies on level-off to distinguish strong cyclic solutions from strong solutions. As defined in [3], strong solutions are guaranteed to achieve the specified goal, while strong cyclic solutions have a chance to terminate and they are guaranteed to achieve the goal state if they terminate.

Before exploiting level-off as discussed above, we need to identify the S-actions and the possibility of obtaining a plan. Thus, another novel concept of FIP is the generation of a weak plan [3] by running the classical Graphplan in the first phase. The plan is weak because it only indicates a possible path to achieve the goal. All the D-actions in the weak plan are identified as S-actions since they generate the intended effects. More importantly, the weak plan represents the optimistic shortest backbone path to the goal because Graphplan always returns the shortest path [1]. Since the search in the second phase is along the shortest path, a substantial performance gain is achieved.

3.1. Phase 1: Weak Plan Generation

In the first phase, the classical Graphplan is run to generate a weak plan, in which all D-actions are S-actions.

Theorem 3.1. At each time step, the F-actions are bypassed by the S-actions and, therefore, cannot be included in the weak plan.

Proof outline: The theorem is based on the fact that the Graphplan algorithm always returns a plan with the shortest path [1]. F-actions that produce failed effects need some remedial actions to correct the effects and, therefore, cannot produce the shortest path.

Consider the tele-control joystick example. The simplified actions are listed in Table 1. The

"ReadJoystick" is a nondeterministic action and is decomposed into two D-actions. To read commands from the joystick, the handle of the joystick device must be acquired. Action "send" is supposed to send the command to the robot through TCP/IP links. Action "CloseJoystick" is responsible for releasing the handle of the joystick when the termination command is issued.

Action	D_ReadJoystick_CTL(LPDIRECTINPUTDEVICE ?joystick, DIJOYSTATE ?js)
Precondition	Acquired(?joystick) & Empty(?js)
Add effect	Avail(?js) & CTLCMD(?js)
Del effect	Empty(?js)
Action	D_ReadJoystick_TM(LPDIRECTINPUTDEVICE ?joystick, DIJOYSTATE ?js)
Precondition	Acquired(?joystick) & Empty(?js)
Add effect	Avail(?js) & TMCMD(?js)
Del effect	Empty(?js) & CTLCMD(?js)
Action	CloseJoystick(DIJOYSTATE ?js, LPDIRECTINPUTDEVICE ?joystick)
Precondition	TMCMD (?js) & Empty(?js)
Add effect	Closed(?joystick)
Del effect	Acquired(?joystick)
Action	Send(DIJOYSTATE ?js)
Precondition	Avail(?js)
Add effect	Empty(?js) & Sent(?js)
Del effect	Avail(?js)

Table 1: Jostick Actions

If the planning problem is that the handle of the joystick device is acquired and the goal is to close the joystick, then the weak plan is as shown in Figure 3, including only the S-action and G-actions.

D_ReadJoystick_TM;
Send;
CloseJoystick;

Figure 3: A weak plan for the tele-control example

This is a weak plan because it does not consider the effect of receiving a normal control command at all. We define the weak plan generated in the first phase as the ***backbone*** weak plan $\boldsymbol{WP_b}$ because subsequent procedural plans are centered on it. We then trim the planning graph by including only actions in the backbone weak plan and the propositions related to these actions. This is to keep only related states and actions. We call the finalized planning graph as $\boldsymbol{WG_b}$.

3.2. Phase 2: Complete Plan Generation

As the backbone weak plan WP_b does not consider failed effects at each time step, the idea of the second phase algorithm is to treat each of the failed effects as a separate planning problem and generate a plan for it. We call this process "***sub-planning***". The novelty is that we manipulate the planning graph so that strong cyclic solutions for sub-plannings end up with level-off while strong solutions, as usual, achieve the goal directly. We can, thus, easily distinguish strong cyclic solutions from strong solutions. Figure 4 illustrates the outline of the algorithm.

```
/* Main Function for the first algorithm
*  Given a planning problem, a backbone weak plan WP_b is generated first.
*  Assumption: Planning domain is dead-end free
*/
Main(WP_b)
1.  Treat each failed effect e with respect to a D-action in WP_b as a planning problem
2.      Construct a planning graph for this planning problem
3.      If level-off Then
4.          Generate a strong cyclic sub-solution;
5.      Else
6.          Generate a sub-solution WP_b';
7.          If there are D-actions involved in WP_b' Then
8.              Main(WP_b');
9.          End If
10.     End If
11. Repeat step 1 until all the failed effects are handled
```

Figure 4: Outline of the algorithm

3.2.1. Sub-planning Graph Expansion. The initial proposition level of the sub-planning for the failed effect e is the proposition level that enables the sub-planning in WG_b. Let the proposition level in WG_b be P_i that enables the sub-planning and let α be the D-action that generates the failed effect e. Then the first action level includes only α.

The purpose of selecting the initial proposition and action levels in this way is two folds. The first goal is to include as much information in WP_b as possible so that the sub-planning will not spend efforts to deal with previously handled states. The second objective is to introduce the failed effect to the sub-planning.

After the initial proposition level is set, the graph expansion phase starts. But it is important to avoid using any other D-actions that originate from the same nondeterministic action as α to expand the planning graph during the sub-planning process. This is to prevent other related D-actions from adversely impacting the sub-planning process.

3.2.2. Level-off Handling. If the sub-planning eventually turns out to be a strong solution, then we add the D-action α to a set $\boldsymbol{C_s}$[1], which will help us to determine the termination point of a loop. It is also possible that the sub-planning graph levels off, in which case it suggests that a strong cyclic solution exists.

[1] By default, the S-action is included in C_s.

Theorem 3.2 During a sub-planning for a failed effect e_f, if the planning graph levels off, then a strong cyclic solution exists in a dead-end free domain.

Proof Outline: Since the domain is dead-end free, there should be at least one path from any reachable state to the goal state. However, level-off does take place and the goal state is not reachable. This fact suggests that there is no direct path leading to the goal state from e_f. To reach the goal state, it must go through a path generated by some related D-actions. In addition, the way of choosing the initial proposition level ensures that all previous states in causal order before the state enabling the sub-planning have been included for the sub-planning. There must be a sequence of actions transiting the state corresponding to e_f to a certain state s_r that appears no later than the state enabling the sub-planning in WG_b. s_r must be included in the initial proposition level so that the planning graph can level off. This typically forms a loop. .

To locate a state in the planning graph, we need to locate the propositions involved in the state. We use the level-membership (*lms*) function introduced in [11] to facilitate the state detection. Let p be a proposition that appears in proposition level i for the first time. Then, $lms(p) = min\{i \mid P_i \text{ contains proposition } p\}$.

Given a state s, function *LevelId* returns the identifier of the first proposition level which contains s.

$LevelId(s) = max(lms(p) \mid p \in s)$

In FIP, *lms* is implemented using a hash table to help with efficient search. Initially, *lms* only contains the information of the propositions in WG_b and then new propositions are appended to *lms* after a sub-plan is generated and its sub-planning graph is finalized. The computation of the appropriate state s_r when level-off takes place proceeds in the following steps as shown in Figure 5.

(1) Let A_l be the last action level and $\wp$ be the set of propositions generated by actions (do not include no-ops) in A_l in the sub-planning graph.

(2) Compute $\langle l_{id}, p \rangle = \langle max(lms(p) \mid p \in \wp), p \rangle$.

(3) Identify state s_r such that $p \in s_r$ and $LevelId(s_r) \geq l_{id}$.

(4) Treat s_r as the goal and do back-chaining search to get a plan.

Figure 5 Computation steps for the repeated state

The idea is to find a proposition p that is generated by some action and is closest to the goal (i.e., steps (1) and (2)). Requiring p to be closest to the goal is to improve the quality of the plan. Then, the state s_r containing p is located as indicated by the location of p. s_r becomes the new goal for the sub-planning.

After a plan is generated, the sub-planning graph is finalized to include only actions in the plan and their related propositions. Then, append the propositions that are not included in WG_b to *lms* and the proposition levels that are after the level where the sub-planning is enabled in WG_b. This operation ensures that the newly handled states in the sub-planning are accessible for later sub-planning problems.

The plan generated in the sub-planning might still be a weak plan, i.e., there might be D-actions involved in it as shown in line 7 in Figure 4. In this case, the previous process is applied recursively to the newly generated weak plan until all the failed effects are handled.

We use the tele-control joystick example to illustrate how the second phase works. Two separate planning graphs are shown in Figure 6. The graph in the left portion is the backbone weak planning graph. The sub-planning for the effect of receiving normal command is shown in the right portion. The initial proposition level is obtained based on P_0 in WG_b and the first action level contains only the D-action, D_ReadJoystick_CTL, as discussed in Section 3.2.1.

The sub-planning graph levels off, which means a strong cyclic solution exists according to Theorem 3.2. Following the steps in Figure 5, the previously handled state included in the last proposition level in the sub-planning graph is identified as "*Empty* $\wedge$ *Acquired*", which is the initial state of the backbone weak plan. Therefore, the goal is set to "*Empty* $\wedge$ *Acquired*" and results in a sub-plan "Send".

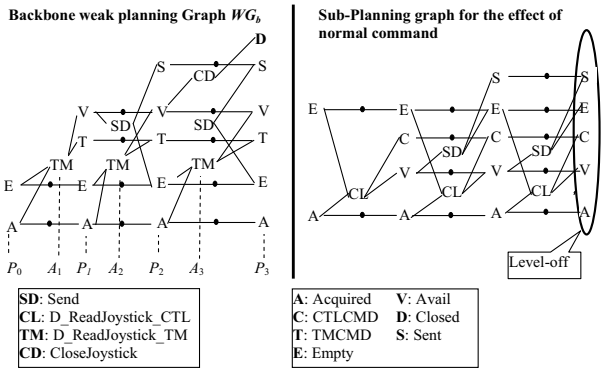

Figure 6: Planning Graph Expansion in the Second Phase

After the planning graph expansion has been completed and the remedial actions for each failed effect have been determined, we can generate loop(s) from the planning graph with the help of the following two Propositions.

Proposition 3.1. A loop starts at the second proposition level of the sub-planning graph which levels off.

Proof outline: Because the sub-planning turns out to be a strong cyclic solution, the application of the F-action is the reason why a loop is formed. Hence, the loop starts from the second proposition level which is generated after the application of the F-action.

Proposition 3.2. Actions in C_s that have a path leading to the given goal represent the exit point of the loop.

Proof outline: Level-off represents an endless loop, which means that there is no direct path from the failed effect to the goal state. Level-off takes place because the related effects are not allowed to be included in the current graph. This fact implies that the path resulting from the failed effect must go through the paths resulting from the related effects to reach the goal state in a dead-end free domain. The actions in C_s can generate such effects and result in paths to the goal.

After the loop boundary is identified, we use the remedial actions for each failed effect to generate an "if" branch. Then, the application of the S-action and other D-actions in C_s after level-off represents the nondeterministic action eventually generating the effects leading to the goal. For example, the sub-plan in Figure 6 is interpreted as "*if* (*CTLCMD*) {*Send*;}". Figure 7 illustrates the final loop construct.

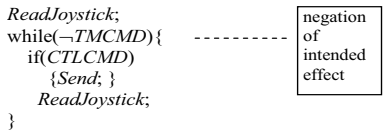

Figure 7: Loop construct for the tele-control example

4. Automated Loop Invariants Detection

A loop invariant is an assertion that is satisfied at the start and at the end of each iteration of a loop and contains important assertions for proving the correctness of the loop upon termination. The proof of the loop invariant includes the following three steps:

- Initialization: The loop invariant must be true before the start of the loop;
- Maintenance: If the loop invariant is true before an iteration of the loop, it must be true before the next iteration;
- Termination: The loop invariant must still hold when the loop terminates.

We present a systematic loop invariant discovery approach by following the above three steps. We define the following notations for specifying the problem:

Let loop L be identified in a sub-planning headed for the failed effect e_f. Let F-action α' generate the failed effect e_f. Let P_i be the proposition level enabling this sub-planning problem in the backbone weak planning graph and let "α_0; α_1; …; α_k" be the sequence of actions before P_i in the backbone weak plan. Then, for initialization, we can obtain a set of propositions that might be the loop invariant for L as follows.

$$LI_L = \gamma(s_\alpha, \alpha') \quad (1)$$

where $s_\alpha = \gamma(\ldots \gamma(\gamma(s_0, \alpha_1), \alpha_2) \ldots \alpha_k)$ is the state enabling α'; s_0 is the initial state of the planning problem; and γ is the state transition function defined in Definition 2.1.

Theorem 4.1. The method given in (1) for choosing LI_L is correct for initialization.

Proof outline: According to Proposition 3.1, the loop starts from the second proposition level P_{s1} of the sub-planning graph. (1) defines the state where the F-action α' is applied. This state is contained in P_{s1} according to Section 3.2.1 and every proposition in the state is true at the moment. Hence, they are possible loop invariants right before the loop starts.

Next, we can determine the ending point of loop L according to Proposition 3.2. Let $a \in C_s$, where actions in C_s can result in paths leading to the goal. Let P_j be the proposition level immediately after the action level where a is located in the planning graph. The set of propositions LI_L obtained in the previous step can be refined as follows:

$$LI_L = LI_L - \{p \mid (p, q) \in \mu P_j \text{ for } p \in LI_L \text{ and } q \in add(a)\} \quad (2)$$

Theorem 4.2. The refinement of LI_L given in (2) is correct for termination.

Proof outline: In order to make LI_L true upon termination of the loop, no proposition in LI_L can be mutex with the add effects of a, where $a \in C_s$ and action a can result in a path leading to the goal.

At this point, LI_L is true before the loop starts and is also true when the loop terminates. We should also make LI_L hold before each iteration of the loop starts and also ensure that it holds before the next iteration starts, i.e., maintenance. For the failed effect e_f, let a_{f1}; a_{f2}; …; a_{fk} be the sequence of remedial actions with a_{f1} being the F-action generating e_f, and s_α be the state enabling a_{f1}. Then LI_L is refined as follows:

$$LI_L = LI_L - \{p \mid (p, q) \text{ is mutex and } p \in LI_L \text{ and } q \in W = \gamma(\ldots \gamma(\gamma(s_\alpha, a_{f1}), a_{f2}) \ldots a_{fk})\} \quad (3)$$

where γ is the state transition function defined in Definition 2.1.

Theorem 4.3. The refinement of LI_L given in (3) is correct for maintenance.

Proof outline: The definition of W in (3) is the final state resulting by the sequence of actions a_{f1}; a_{f2}; …; a_{fk}. In order to make LI_L true before the next iteration, no propositions in LI_L could be mutex with a proposition in W. Since LI_L is true before the first iteration according to Theorem 4.1, by induction we can prove that it will be true before every iteration by following the updating rule of (3).

For the tele-control example, as shown in Figure 6, we can obtain the initial value of LI_L according to (1), i.e., $LI_L = \gamma(s_0, CL) = \{V, C, A\}$. According to (2), $LI_L = \{V, C, A\} - \{C\} = \{V, A\}$ because "*C*" represents "*CTLCMD*" which is mutex with the intended effect of "*TMCMD*". Finally, according to (3), $LI_L = \{V, A\} - \{V\} = \{A\}$. This is because $\gamma(\gamma(\{E, A\}, CL), SD) = \{S, C, E, A\}$ and "V" represents "*Avail*" that is mutex with "E" that denotes "*Empty*". Hence, the final loop invariant is {A}. Obviously, this is an accurate and meaningful result because the joystick will not work if the handle of the device is lost.

4.1. Refining Invariants

The approach presented above may include propositions that are independent of the concerned loop as part of the loop invariant. For example, some propositions that appear in the levels before the loop starts may be generated by some actions. They may be irrelevant to the loop under analysis. Therefore, they may hold for (1), (2), and (3) and the proposed method would recognize them as part of the loop invariants.

To avoid this situation, we use the following heuristic method to refine the loop invariant.

Let Obj_a = {the set of objects involved in instantiating the F-actions and their corresponding S-action} and let Obj_p = {the set of objects involved in instantiating the proposition p}. If $Obj_a \cap Obj_p = \phi$, then report proposition p to be a possibly spurious invariant.

4.2. Combining with other Techniques

The completeness of the discovered loop invariants depends highly on how the planning domain is created and how the planning problems are given. The proposed approach cannot discover loop invariants if the possible invariants are not modeled as propositions. For example, a carefully conceived planning domain can be created to solve the GCD (Greatest Common Divisor) problem [6]. The generated plan is as follows:

```
Compare(x, y);
while (not Equal(x,y)){
   if (Greater(y, x)){  Deduct (y, x);}
   else if (Greater(x, y)){  Deduct (x, y);}
   Compare(x, y );}
```

Here, "*Deduct*(*x*, *y*)" means "*x* – *y*". As "*Compare*(*x*, *y*)" can be issued anytime as long as two positive integers are available, the planning problem does not provide much extra information for discovering the loop invariants. Daikon [4] is a well-known tool that is capable of detecting loop invariants involving range limits. It can detect the loop invariant of the GCD problem by reporting the range limits, i.e., $x > 0$ and $y > 0$. Of course, if the planning problem can present Greater(x, 0) & Greater(y, 0) in the initial condition, the proposed approach can discover the loop invariant correctly. But this example shows that by combining the above approach with other loop invariant detection techniques, it can produce even more accurate and meaningful loop invariants.

5. Related Works

In 1990s, deductive program synthesis methods thrived and were once regarded as the key to achieving automated program synthesis [14]. KIDS [19] and AMPHION [21] are two successful representatives of this approach. The advantage of deductive synthesis is that the generated programs are correct-by-construction. However, these general-purpose systems are either not very efficient or not very powerful in generating complex programs. Specware [20] extends KIDS and employs stepwise refinement to transform formal specifications into executable codes. Planware [1] is an extension of Specware with focus on generating high-performance schedulers from specifications of scheduling problems. Planware supports automatic construction of a domain theory for a particular scheduling problem. Its performance is much better than that of KIDS. But the cost is that it only supports sharply restricted domains [1].

Based on AMPHION, NASA developed domain specific generators that separate verifications from program generation. For example, loop invariants are used to verify the correctness of the generated Kalman Filters in [8].

Loop invariants play an important role in specifying the behaviors of programs that include loops. Loop invariants are helpful in both the initial system development activities as well as subsequent system maintenance tasks. Being aware of loop invariants, developers are unlikely to violate the properties that must be preserved in a program.

Daikon [4] is a well known tool for dynamically detecting program invariants. It infers possible invariants in a program by really running the program. First, the interested variables are identified and instrumented. Then, a set of test cases is collected, over which the program runs. Finally, invariants that behave consistently during the tests are reported to the programmer.

Daikon suffers from the problem of reporting too many invariants, some of which may be spurious. The work in [10] proposed a new structural coverage criterion to improve the accuracy of the dynamically detected invariants. Daikon is executed first to obtain a set of likely invariants. Then, the invariant-coverage suites for these likely invariants are generated. After Daikon runs over the suites, some of the false invariants are removed.

All dynamic approaches suffer from the inherent problem that the quality of the detected invariants relies substantially on the quality and completeness of test cases [4]. They are not theoretically sound. On the other hand, theorem proving is extensively used to facilitate the loop invariant detection. In [13], the loop-invariant computations are initiated only when there is a need for stronger loop invariant. Hence, this is an iterative process by first generating the verification condition which is sent to a theorem prover to prove its validity. If it fails, the set of candidate traces responsible for the failure are passed to an abstract interpreter on the loops so that it might find stronger loop invariants that allow the theorem prover to make more progress toward a proof. Although this is a sound technique, its scalability and the degree of easy-to-use still remain unclear.

Loop invariants are also important for program synthesis because of the possibility that some latent defects in the program synthesis tool may result in the generation of incorrect code. Hence, they use verification technique to guarantee the correctness [8]. As loop invariants are indispensable in automated program verification, the ability to automatically detect loop invariants is very critical. Our approach is sound and can ensure the validity of the detected loop invariants, and, therefore, greatly facilitates high-confidence system verification techniques.

6. Conclusions and Future Work

In this paper, we have presented an automated loop invariant discovery approach via planning graph analysis. Loop invariants can be discovered as a by-product of the planning process. Specifically, after a plan is generated, the final planning graph can be used for loop invariants discovery. Two propositions are presented to help identify the loop boundaries. Then, loop invariants are computed in three steps, i.e., initialization, maintenance, and termination, which are standard proof steps for validating loop invariants. The proposed approach has rigorous theoretical basis and is sound for loop invariant discovery. Although the focus of the paper is on loop invariants, the proposed approach can help discover all kinds of program invariants by following the same approach.

As the proposed approach exploits only Graphplan's intrinsic features, it can be seamlessly applied to Graphplan's large number of variants, such as the well-known FF [11]. As shown in Figure 1, a hierarchical invariants discovery structure can be formed with the proposed approach addressing the invariant discovery for the glue code and leveraging the capabilities of other invariant detection techniques, e.g., Daikon [4], as needed to identify appropriate loop invariants for the underlying components. This can greatly facilitate high-confidence system verification process and also improve the system's reliability. In addition, AI planning is extensively used in service-oriented composition and dynamic workflow generation. The proposed approach can be applied to this area also.

Some future research directions include further analysis of the informative structure of planning graphs and use of the loop invariants to verify the corresponding programs. Planning graphs can also provide more information for discovering other types of invariants, such as class and service invariants, for comprehensive system verification.

7. References

[1] Blaine, L., Gilham, L., Liu, J., Smith, D.R., and Westfold, S. "Planware -- Domain-Specific Synthesis of High-Performance Schedulers", Proceedings of the Thirteenth Automated Software Engineering Conference, IEEE Computer Society Press, Los Alamitos, CA, pp. 270-280.

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

[3] Cimatti, A.; Pistore, M.; Roveri, M.; and Traverso, P. 2003. "Weak, strong, and strong cyclic planning via symbolic model checking," *Artificial Intelligence*, 147(1–2):35–84.

[4] Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D. 2001. "Dynamically discovering likely program invariants to support program evolution," *IEEE Trans Softw Eng* 27(2):1–25

[5] Fu, J., Bastani, F. B., and Yen, I. 2006. "Automated AI planning and code pattern based code synthesis," *ICTAI* 2006, pp. 540–546.

[6] Fu, J., Bastani, F. B., and Yen, I. 2007. "Iterative planning in the context of automated code synthesis," *COMPSAC* 2007: 251–259.

[7] Fu, J., Bastani, F. B., Ng, V., Yen, I., and Zhang, Y. 2008. "FIP: A fast planning–graph–based iterative planner," *Technical Report*, *UTDCS*–03–08, *UT–DALLAS*.

[8] Gamboa, R., Cowles, J., Baalen, J.V. 2003. "On the verification of synthesized Kalman filters," 4*th International Workshop on the ACL*2 *Theorem Prover and Its Applications*.

[9] Gazen, C.; Knoblock, C. A. 1997. "Combining the expressivity of UCPOP with the efficiency of Graphplan," In *Proceedings of ECP*–97, 221–233.

[10] Gupta, N. and Heidepriem, Z.V. 2003. "A new structural coverage criterion for dynamic detection of program invariants," In *Proc*. 18*th IEEE International Conference on Automated Software Engineering*, pp. 49–58

[11] Hoffmann J. and Nebel B. 2001. "The FF planning system: Fast plan generation through heuristic search," in *Journal of Artificial Intelligence Research*, Volume 14, 253–302.

[12] Kuter, U. 2004. "Pushing the limits of AI planning," In *Proc*. *of the Doctoral Consortium* at the 14*th International Conference on Automated Planning and Scheduling* (*ICAPS*–04).

[13] Leino K. and Logozzo F. 2005. "Loop invariants on demand," *APLAS* 2005, *LUNCS* 3780, pp. 119–134.

[14] Loveland, D. W. 2000. "Automated deduction: achievements and future directions", *Commun*. *ACM*, 43, 11es (Nov. 2000), 10.

[15] Malik Ghallab, Dana Nau, Paolo Traverso, *Automated Planning*: *Theory and Practice*, Morgan Kaufmann (2004), pp. 113–139.

[16] McDermott D., et al. 2004. "The PDDL Planning Domain Definition Language," *The AIPS*–2004 *Planning Competition Committee*.

[17] Mitra, D. and Bond, W.P. 2002. "Component–oriented programming as an AI–planning problem", In *Proc*. *of the* 15*th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*: *Developments in Applied Artificial Intelligence*, pp. 567–574.

[18] Rao J. and Su X. 2004. "A survey of automated web service composition methods," In *Proc*. *of the First International Workshop on Semantic Web Services and Web Process Composition*, *SWSWPC* 2004, San Diego, California.

[19] Smith, D. R. 1990. "KIDS: A Semiauomatic Program Development System". *IEEE Transactions on Software Engineering*. VOL. 16, NO. 9. 1990.

[20] Srinivas, Y. V. and Jullig, R. 1995. Specware: Formal support for composing software. In B. Moeller, editor, Proceedings of the Conference on Mathematics of Program Construction, pages 399–422. LNCS 947, Springer-Verlag, Berlin.

[21] Stickel, M., Waldinger, R., Lowry, M., Pressburger, T., and Underwood, I. 1994. "Deductive composition of astronomical software from subroutine libraries," *Proceedings* 12*th International Conference on Automated Deduction* (*CADE*–12), Nancy, France.

[22] Yen, I., Bastani, F., Mohamed, F., Ma, H., and Linn, J. 2002. "Application of AI planning techniques to automated code synthesis and testing," *Proceedings of the* 14*th IEEE International Conference on Tools with Artificial Intelligence* (*ICTAI*'02), pp. 131–137.