CEUR-WS.org/Vol-2987/paper3.pdf

Automated Planning Through Program Verification

Salvatore La Torre[1], Gennaro Parlato[2]

[1]University of Salerno, Italy
[2]University of Molise, Italy

Abstract
In this paper, we report on a preliminary study on the feasibility of applying techniques and tools for finding errors in programs, or prove them entirely correct, to effectively explore the large state space of instances of the automated planning problem (AP). To leverage the recent advancements in the symbolic program analysis, we design a simple reduction from AP to the configuration reachability problem of programs and then use off-the-shelf program verification tools. We evaluate the feasibility of our approach on the Agricola-sat18 benchmark used at IPC'18.

Keywords
Automated Planning, Formal Methods, Program Verification

1. Introduction

The automated planning problem is a central problem in AI which concerns with the search and the synthesis of a sequence of actions aimed to reach a given goal. It is a complex and well-studied problem, and in the years several efficient solutions have been proposed in the literature to solve it. These include direct approaches such as forward or backward chain searches and partial order scheduling [1]. Other solutions consist of reducing it to other problems for which scalable and effective solutions exist, such as Boolean formula satisfiability (SAT) or model checking [2, 3].

In this paper, we expand this arsenal of solutions by contributing another reduction, this time to the program verification problem. Given an instance of the planning problem, we construct a simple imperative program that nondeterministically simulates a sequence of actions starting from an initial state that fails an assertion whenever a target state is reached (see [4, 5, 6, 7, 8, 9] for similar approaches in other domains). This type of reduction, although simple, opens up the possibility of using off-the-shelf automatic techniques and tools designed to verify programs in order to synthesise plans. These include approaches such as Bounded Model Checking (BMC), Abstract Interpretation, Counter-example Guided Abstraction Refinement, to name a few (see [10]). The programs produced through our reduction have a very particular form that we think could be exploited to refine and then specialise these approaches and techniques to work well for this class of noncanonical programs.

OVERLAY 2021: 3rd Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis, September 22, 2021, Padova, Italy

slatorre@unisa.it (S. La Torre); gennaro.parlato@unimol.it (G. Parlato)
0000-0002-4978-4307 (S. La Torre); 0000-0002-8697-2980 (G. Parlato)

© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

We begin our study following this direction, exploring the possibility of using BMC-based techniques in a simple way. We apply our approach to some benchmarks taken from the planning competition to demonstrate its feasibility. We leave for future investigations the possibility of exploiting other program verification techniques and tools.

2. Preliminaries

The planning problem is essentially a search problem over the states of a transition system, a set of states along with a set of actions that can change the current system state. Thus, given a set of states S and a set of actions A over them, the *planning problem* simply asks whether there is a sequence of actions from A that starting from an *initial state* take the system to a state in a *goal* set G.

```
instance ::= domain problem
domain   ::= (define (domain id)
                (:constants idlist)
                (:predicates plist)
                actions             )
actions  ::= actions actions |
             (action id
                (:parameters parlist)
                (:precondition b)
                (:effect b)         )
problem  ::= (define (problem id)
                (:domain id)
                (:objects idlist)
                (:init atlist)
                (:goal b)           )
```

Figure 1: Planning language.

To express the planning problem instances, we consider a simple PDDL-like language. The syntax is given in Figure 1. Here, an *instance* is composed of a domain and a problem. The *domain* part essentially identifies the transition system by defining the possible states through a set of Boolean *predicates* and *constants*, and a set of *actions*. Each action is defined over a list of parameters (that can assume the values of the given constants) and has a *precondition* and an *effect* with the meaning that an action can be taken on each state where the precondition holds and when taken, produces the change of the truth values of the predicates as described in the effect part. The *problem* part instead completes the domain with more constants (*object* part), and gives the initial values for the predicates (identifying the initial state) and the condition that must old for the goal states.

The syntax allowed for the preconditions and goals may vary depending on the specific planning language. Also, constants and objects may be typed, and functions that manipulate numeric types can be added. To keep the presentation simple here we limit to conditions that are just conjunctions of positive and negative atoms, and omit functions and types. However, these features and more complex conditions can be easily included in our code-to-code transformation.

3. Reduction to program verification: code-to-code translation

In this section, we describe the code-to-code translation that is the main part of our reduction. Instead of giving the formal translation we illustrate it by an example. For this we will use a classical and well-known planning domain, the so-called *blocks world*.

Example: blocks world. The blocks world domain consists of a set of cube-shaped blocks sitting on a table. The blocks can be piled up such that only one block can fit directly on top of another. A robot can pick up a block that is not below other ones (*top*) and move it to another position, either on the table or on top of another block. In this domain, both the initial state and the goal may consist of one or more piles of blocks.

Figure 2.(a) gives a PDDL-like encoding of a planning instance based on this domain where the initial state is given by two piles: one formed by the sole block `B` and the other formed by block `C` on top of block `A`. The goal instead consists of a single pile formed by block `A` on `B` and block `B` on `C`. We use the predicate `On(x,y)` to indicate that block `x` is on `y` (note that parameters are preceded by `?` in the style of PDDL-syntax), where `y` is either another block or the `table`. We use another predicate `Clear(x)` to denote that `x` is top. In the domain, the actions are `move` and `moveToTable`. Action `move(b,x,y)` moves a block `b` on top of a block `y` provided that `b` and `y` are both top, additionally the object `x` (that might be a block as well or the `table`) sitting right below `b` must become top. Action `moveToTable(b,x)` just moves a top block `b` on the `table` and makes the object `x` below `b` a top one. A solving plan for the described planning instance consists of three steps: `movetoTable(C,A)`, `move(B,table,C)`, and `move(A,table,B)`.

Code-to-code translation. To illustrate our code-to-code translation, we refer to the example shown in Figure 2 (a). We use a simple imperative programming language (such as that of the analyzer ConcurInterproc: http://pop-art.inrialpes.fr/interproc/concurinterprocweb.cgi). The program is shown in Figure 2 (b). We use scalar variables or arrays with global scope to encode predicates. We initialize these variables using the expression `initial` derived from the `init` component of the problem definition. The actions are each modelled with a procedure of the same name. Here we declare a number of local variables that model the parameters. These variables are initialized with a non deterministic value taken from the domain of the variables using the expression `rand`. To simulate the action we first check that the guessed values make the precondition evaluate to `true`, and if so we update the arrays modelling the predicates with a sequence of assignments derived from the `effect` of the action. The simulation is then orchestrated by the main procedure: it goes through an infinite loop whose body is crafted to simulate all actions in a non-

```
(define (domain blocks)
 (:constants table)
 (:predicates (On ?x ?y)(Clear ?x))
 (:action move
  :parameters (?b ?x ?y)
  :precondition (and (On ?b ?x)
            (Clear ?b) (Clear ?y))
  :effect (and (On ?b ?y) (Clear ?x)
    (not(On ?b ?x)) (not(Clear ?y))))
 (:action moveToTable
  :parameters (?b ?x)
  :precondition(and(On?b?x)(Clear?b))
  :effect (and(On ?b table)(Clear ?x)
            (not(On ?b ?x)) )    ))

(define (problem p1)
 (:domain blocks)
 (:objects A B C)
 (:init (On A table) (On B table)
      (On C A) (Clear B) (Clear C) )
 (:goal (and (On A B) (On B C)) )    )
```

(a) PDDL-like instance

```
typedef obj = enum {table,A,B,C};

var On[obj,obj]:bool, Clear[obj]:bool;

initial( On[A,table] == true and
         On[B,table] == true and
         On[C,A] == true     and
         Clear[B] == true    and
         Clear[C] == true        );

proc move()   var b:obj, x:obj, y:obj;
begin
 b=rand; x=rand; y=rand;
 if (Clear[y] and Clear[b] and On[b,x])
 then
   Clear[x] = true; Clear[y] = false;
   On[b,y] = true; On[b,x] = false;
end

proc moveToTable()  var b:obj, x:obj;
begin
 b=rand; x=rand;
  if (On[b,x] and Clear[b]) then
   Clear[x]    = true;
   On[b,table] = true;
   On[b,x]     = false;
end

begin    //main procedure
 while (true)
  if (brand) then move(); endif;
  if (brand) then moveToTable(); endif;
  assert(not On[A,B] or not On[B,C]);
end    //fail if goal conf is reached
```

(b) Program encoding

Figure 2: **(a)** PDDL-like Example **(b)** a program that simulates the behaviour of (a).

deterministic way. The body also contains an assertion whose condition corresponds to the negation of the goal condition of the problem. Thus, to synthesise a plan we check whether the program fails this assertion. If so, by inspecting the counterexample we are able to construct a plan by listing the actions simulated along the counterexample. Of course, an instance of the planning problem that does not admit a plan leads to a program that is actually correct, that is, a program that has no executions leading to an assertion violation.

4. Implementation and experiments

To evaluate our approach, we implemented it into a prototype tool and conducted some preliminary experiments on the Agricola-sat18 benchmark taken from the 9th International Planning Competition (IPC'18) held at ICAPS 2018.

Prototype tool. Our tool is a code-to-code translation from PDDL planning instances to C programs. It is entirely written in python (version 3.8) and relies on the library `pddlpy` [11] to parse the PDDL instances. `pddlpy` provides a convenient API to extract the different kinds of elements of PDDL domains and problems.We have also extended the API of `pddlpy` to simplify the translation into a C program. The prototype uses CBMC (https://www.cprover.org/cbmc/) as backend for the program analysis. Thus a main parameter in the implemented approach is the number of *rounds* where we select the actions (which corresponds to the unwinding depth of the infinite loop of the main procedure). We support also a few more input parameters that can be given to trigger a swarm-like analysis (see [12, 13]), enable some light partial order reductions, and few more search heuristics.

Agricola-sat18. This benchmark set is based on the board game Agricola that models a farm with some workers. The game has a number of turns and stages, in which the player must select actions for the workers that are finalized to obtain more resources. The player may decide also to increase the number of workers. To reach the goal the player may take several actions per turn however this also increases the amount of food consumed at the end of the turn, that may lead into dead ends. The benchmark is composed of twenty planning instances sharing a common domain file. The model is written in the STRIPS fragment of PDDL which is slightly more expressive than the fragment presented in this paper: objects and constants are typed and also cost functions and numerical types are allowed.

Experiments. We run our tool on the entire benchmark set with round bound 20 finding plans for 6/20 problems and taking overall about 900s. We repeated the same experiment with bound 30, now taking about 7000s (including three 1200s timeouts) and finding plans for six more problems. We then focused only on the remaining eight unsolved problems by using increasing number of rounds up to 70 and timeouts up to 7200s. We found plans in three more problems for a total of 15/20 problems. Interestingly, the new plans where discovered with relatively low computational resources: 26 rounds and 1800s timeout, 31 rounds and 3000s timeout, 33 rounds and 300s timeout, respectively. These preliminary experiments show that our approach, although straightforward, is competitive with the best performing tools at the ICP'18 which were only able to solve one more problem, thus confirming our intuition that program verification can play a role in the automated planning domain.

References

[1] M. Ghallab, D. S. Nau, P. Traverso, Automated Planning and Acting, Cambridge University Press, 2016.

[2] J. Rintanen, Planning and SAT, in: A. Biere, M. Heule, H. van Maaren, T. Walsh (Eds.), Handbook of Satisfiability, volume 185 of *Frontiers in Artificial Intelligence and Applications*, 2009, pp. 483–504. URL: https://doi.org/10.3233/978-1-58603-929-5-483.

[3] F. Giunchiglia, P. Traverso, Planning as model checking, in: Recent Advances in AI Planning, Berlin, Heidelberg, 2000, pp. 1–20.

[4] A. L. Ferrara, P. Madhusudan, G. Parlato, Security analysis of role-based access control through program verification, in: 25th IEEE Computer Security Foundations Symposium, CSF 2012, 2012, pp. 113–125. URL: https://doi.org/10.1109/CSF.2012.28.

[5] A. L. Ferrara, P. Madhusudan, G. Parlato, Policy analysis for self-administrated role-based access control, in: Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013. Proceedings, volume 7795 of *LNCS*, 2013, pp. 432–447. URL: https://doi.org/10.1007/978-3-642-36742-7_30.

[6] A. L. Ferrara, P. Madhusudan, T. L. Nguyen, G. Parlato, Vac - verifier of administrative role-based access control policies, in: Computer Aided Verification - 26th International Conference, CAV 2014. Proceedings, volume 8559 of *LNCS*, 2014, pp. 184–191. URL: https://doi.org/10.1007/978-3-319-08867-9_12.

[7] O. Inverso, E. Tomasco, B. Fischer, S. La Torre, G. Parlato, Bounded model checking of multi-threaded C programs via lazy sequentialization, in: Computer Aided Verification - 26th International Conference, CAV 2014. Proceedings, volume 8559 of *LNCS*, 2014, pp. 585–602. URL: https://doi.org/10.1007/978-3-319-08867-9_39.

[8] E. Tomasco, O. Inverso, B. Fischer, S. La Torre, G. Parlato, Verifying concurrent programs by memory unwinding, in: Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015. Proceedings, volume 9035 of *LNCS*, 2015, pp. 551–565. URL: https://doi.org/10.1007/978-3-662-46681-0_52.

[9] T. L. Nguyen, B. Fischer, S. La Torre, G. Parlato, Lazy sequentialization for the safety verification of unbounded concurrent programs, in: Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016. Proceedings, volume 9938 of *LNCS*, 2016, pp. 174–191. URL: https://doi.org/10.1007/978-3-319-46520-3_12.

[10] E. M. Clarke, T. A. Henzinger, H. Veith, R. Bloem (Eds.), Handbook of Model Checking, 2018. URL: https://doi.org/10.1007/978-3-319-10575-8.

[11] H. Foffani, A python PDDL parser, 2017. URL: https://pypi.org/project/pddlpy/.

[12] G. J. Holzmann, R. Joshi, A. Groce, Swarm verification techniques, IEEE Trans. Software Eng. 37 (2011) 845–857. URL: https://doi.org/10.1109/TSE.2010.110.

[13] E. Tomasco, B. Fischer, S. La Torre, T. L. Nguyen, G. Parlato, P. Schrammel, Parallel bug-finding in concurrent programs via reduced interleaving instances, in: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, 2017, pp. 753–764. URL: https://doi.org/10.1109/ASE.2017.8115686.