Discovering and Proving Invariants in Answer Set Programming and Planning

Patrick Lühne

University of Potsdam, Germany,
`patrick.luehne@cs.uni-potsdam.de`

Abstract. Answer set programming (ASP) and planning are two widely used paradigms for solving logic programs with declarative programming. In both cases, the quality of the input programs has a major influence on the quality and performance of the solving or planning process. Hence, programmers need to understand how to make their programs efficient and still correct. In my PhD studies, I explore how input programs can be improved and verified automatically as a means to support programmers. One of my research directions consists in discovering invariants in planning programs without human support, which I implemented in a system called *ginkgo*. Studying dynamic systems in greater depth, I then developed *plasp* 3 with members of my research group, which is a significant step forward in effective planning in ASP. As a second research direction, I am concerned with automating the verification of ASP programs against formal specifications. For this joint work with Lifschitzs group at the University of Texas, I currently develop a verification system called *anthem*. In my future PhD studies, I will extend my research concerning the discovery and verification of ASP and planning problems.

1 Introduction

Answer set programming (ASP) and planning are two widely used paradigms for solving logic programs with declarative programming. While ASP aims to be as general-purpose as possible, planning focuses on dynamic systems, where sequences of actions are searched in order to achieve specific goals. As with other knowledge representation paradigms, quality and performance in solving and planning not only depend on the implementations of solvers and planners but also on the quality of the input program specifications. From the perspective of programmers working with solvers and planners, it is, hence, fundamentally important to understand whether their programs are efficient and, more importantly, correct with respect to their specifications.

In my PhD studies, I explore how input programs can be improved and verified automatically as a means to support programmers. Both of these objectives relate to *invariants* of logic programs—properties that preserve the correctness of a program.

My first research direction consists in *discovering* invariants without human support. For this purpose, I developed the system *ginkgo*, which continuously discovers invariants in planning problems by generalizing conflict constraints learned by an ASP solver (Section 2). While working with planning problems in the *planning domain definition language* (PDDL [8]), I further implemented *plasp* 3, the third generation of an ASP planning system. With *plasp*, PDDL programs can be solved with established ASP solvers such as *clingo*. Based on this effort, members of our research group and I showed how to make planning in ASP more effective with parallel planning (Section 3).

As a second research direction, I investigate how to automate the *verification* of ASP programs against formal specifications in a subset of the input language of *clingo* in cooperation with Vladimir Lifschitzs group at the University of Texas. *anthem*, another system that I currently develop, aims to perform this task by translating ASP programs to first-order logic formulas to validate them against a specification by a theorem prover (Section 4).

In future work, I will extend my research concerning the discovery and verification of ASP and planning problems. Section 5 discusses such directions, before Section 6 concludes this extended abstract.

2 *ginkgo*Discovering Invariants in ASP Planning

Conflict learning has become a base technology in Boolean constraint solving, and, in particular, answer set programming. However, learned constraints are only valid for a currently solved problem instance and do not carry over to similar instances. To address this issue, I developed a framework featuring an integrated feedback loop that allows for reusing conflict constraints [3]. The idea is to extract (propositional) conflict constraints, generalize and validate them, and reuse them as integrity constraints. In this way, an input program is continuously extended with automatically discovered invariants. Although I explored this approach in the context of dynamic systems (specifically, PDDL planning), the ultimate objective is to overcome the issue that learned knowledge is bound to specific problem instances.

I implemented this workflow in two systems, namely, a variant of the ASP solver *clasp* that extracts integrity constraints, along with the downstream system *ginkgo*[1] for generalizing and validating them. *ginkgo* finds invariants by first deriving candidate properties (learned constraints that are generalized over the temporal domain). These properties are then checked for invariance. This relies on automated proofs that I fully implemented in ASP with meta encodings.

3 *plasp* 3Towards Effective ASP Planning

Emerging from my work with ASP-based planning in the *ginkgo* system, I implemented the third installment of *plasp*[2]. While earlier versions of *plasp* were pure PDDL-to-ASP translators [4], *plasp* 3 was conceived to provide a flexible

[1] https://github.com/potassco/ginkgo
[2] https://github.com/potassco/plasp

platform to experiment with a variety of techniques to make planning in ASP more effective (to be published at LPNMR 2017 [2]).

For this purpose, I reimplemented *plasp*, while widening the range of accepted PDDL features in comparison to the previous versions. Further, our research group developed novel planning encodings, some inspired by SAT planning and others exploiting ASP features such as well-foundedness. I designed *plasp* 3 such that it handles multivalued fluents and, hence, captures both PDDL as well as SAS planning formats. Third, enabled by multishot ASP solving, advanced planning algorithms are offered, also borrowed from SAT planning. Empirical analyses show that these techniques have a significant impact on the performance of ASP planning.

4 *anthem*Verifying the Correctness of ASP Programs

In their recent work (to be published [5,6]), Harrison, Lifschitz, and Raju have extended the definition of program completion to a subset of the input language of *clingo*. The aim of their work is to extend the applicability of formal verification methods to ASP by turning logic programs into completed definitions. This can also be understood as a translation from *clingo*s input language to first-order logic formulas.

In cooperation with their research group at the University of Texas, I started developing a system called *anthem*,[3] which performs the completion of logic programs automatically. After translating and simplifying formulas with *anthem*, programmers can see more clearly what exactly their program solves.

Furthermore, the first-order logic representation allows us to perform automated proofs by employing established theorem provers, which commonly operate on similar input formats. Popular theorem provers include *E* [10], *Coq* [1], and *Prover9* [7]. Our goal is to extend *anthem* such that it can be used to quickly test whether ASP programs fulfill given invariants. With such a tool, programmers could start writing programs by first making a formal specification, against which their code is later verified.

5 Future Work

As stated before, my most recent work focuses on using theorem provers to verify that logic programs comply with a given specification. This indirection of proving invariants through first-order logic might turn out particularly useful when coming back to my earlier research on the *ginkgo* system. This is because there are many established first-order theorem provers, which might make for a stronger proof system than the counterexample-based validation method currently employed by *ginkgo*.

Furthermore, I will expand my research in the field of PDDL planning. Building on *plasp* 3, I want to explore how far planning in ASP can be pushed, with

[3] https://github.com/potassco/anthem

the objective of achieving performance on par with state-of-the-art SAT planners such as *Madagascar* [9].

Finally, I am always exploring opportunities to apply the planning-related techniques to the broader scope of general ASP programs. This involves the research areas of automatic modeling, program synthesis, and superoptimization, which I want to further familiarize myself in my upcoming PhD studies.

6 Conclusions

Concerning the automatic discovery and verification of invariants in logic programs, I already made insightful progress. In my early PhD projects, I showed the feasibility of reusing learned conflict constraints in ASP planning by means of generalization with my *ginkgo* system. I further studied dynamic systems as such and helped making ASP planning much more effective with *plasp* 3 and performance-wise closer to state-of-the-art SAT planners than previous attempts. I believe that these two systems I developed could make use of refined invariant finding techniques, which is one of the things I want to study in the remainder of my PhD studies.

Furthermore, I am researching automated verification techniques in multiple contexts. First, as a means to validate potential candidate invariants within *ginkgo*. Second, to completely automate the process of testing ASP programs against formal specifications, which is the objective of my currently work-in-progress system *anthem*. This is a technique that could later be useful for other parts of my research as well.

To my mind, there are many interesting aspects of discovering and verifying invariants ahead that I want to address in my PhD studies. This also includes more practical applications such as making ASP planning yet more effective.

References

1. Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. The Coq proof assistant reference manual: Version 8.6. https://coq.inria.fr/distrib/current/refman/, 2016.
2. Y. Dimopoulos, M. Gebser, P. Lhne, J. Romero, and T. Schaub. *plasp* 3: Towards effective ASP planning *(to appear)*. In *Proceedings of the Fourteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'17)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 2017.
3. M. Gebser, R. Kaminski, B. Kaufmann, P. Lhne, J. Romero, and T. Schaub. Answer set solving with generalized learned constraints. In M. Carro and A. King, editors, *Technical Communications of the Thirty-second International Conference on Logic Programming (ICLP'16)*, volume 52, pages 9:1–9:15. Open Access Series in Informatics (OASIcs), 2016.
4. M. Gebser, R. Kaminski, M. Knecht, and T. Schaub. plasp: A prototype for PDDL-based planning in ASP. In J. Delgrande and W. Faber, editors, *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic*

Reasoning (LPNMR'11), volume 6645 of *Lecture Notes in Artificial Intelligence*, pages 358–363. Springer-Verlag, 2011.

5. Amelia Harrison, Vladimir Lifschitz, and Dhananjay Raju. Program completion in the input language of gringo. Submitted for publication, 2017.
6. Vladimir Lifschitz. Achievements in answer set programming (preliminary report). In *Working Notes of the Workshop on Answer Set Programming and Other Computing Paradigms*, 2016.
7. W. McCune. Prover9 and Mace4. http://www.cs.unm.edu/~mccune/prover9/, 2005–2010.
8. D. McDermott. PDDL—the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.
9. J. Rintanen. Madagascar: Scalable planning with SAT. In M. Vallati, L. Chrpa, and T. McCluskey, editors, *Proceedings of the Eighth International Planning Competition (IPC'14)*, pages 66–70. University of Huddersfield, 2014.
10. Stephan Schulz. System Description: E 1.8. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*. Springer, 2013.