A Case for Query-driven Predicate Answer Set Programming (Position Paper)

Gopal Gupta
Elmer Salazar, Kyle Marple, Zhuo Chen, Farhad Shakerin

Department of Computer Science
The University of Texas at Dallas

1 Answer Set Programming

Answer Set Programming (ASP) [1] has emerged as a successful paradigm for developing intelligent applications. ASP is based on adding *negation as failure* to logic programming under the stable model semantics regime [2]. ASP allows for sophisticated reasoning mechanisms that are employed by humans (and that can be clubbed under the rubric of common sense reasoning) to be modeled elegantly. The main characteristic of answer set programming is that it supports non-monotonic logic-based reasoning mechanisms that can model default reasoning, counterfactual reasoning, abductive reasoning, etc.—mechanisms that humans use for everyday reasoning.

ASP is particularly good at modeling reasoning based on default rules (and exceptions to these default rules) [1]. Humans simplify their burden of reasoning by relying on default rules. For example, we will automatically assume that if Tweety is a bird, it flies. In general, we discount the possibility that Tweety might be a penguin, an ostrich, or it may be wounded, etc. We may not consider these exceptional situations right in the beginning. However, once we learn that that one of the exceptional situation holds, we will withdraw our earlier conclusion that Tweety can fly. Likewise, if I call a very close friend of mine on his mobile phone in the morning on a weekday, and he doesn't pick it up, then I would immediately conclude that he is busy getting ready for work (because normally he picks up the phone immediately). We will discount less likely reasons such as he may be sick and may be in the hospital or may have had to leave for work early, unless we learn that that is indeed the case. In fact, I may take further actions based on the default conclusion I reached (e.g., I need to talk to my friend urgently, so next I will call his home phone, because I have concluded that he is home, he is just not picking up his mobile phone).

There are many other such reasoning patterns that humans use that can be elegantly modeled by ASP as well, e.g., preferences, anti-recommendation, concomitant choice, indispensable choice, incompatible choice, etc. [8]. One could argue that for many applications, such as self-driving cars, human-styled common sense reasoning is a must. Arguably, ASP should be sufficient for modeling human-style reasoning. That is because well-founded reasoning (inductive reasoning) as well as cyclical reasoning (establishing mutual consistency or *coinductive* reasoning) can be both represented in ASP, permitting a wide-range of reasoning patterns to be modeled [7]. Our experience modeling a physician advisory system for chronic heart failure indeed bears this out [8].

ASP in our opinion represents "usable automated reasoning" mentioned in the call for papers of this workshop. An important characteristic of ASP is that each rule is easy to understand in isolation. However, it is hard to understand what these rules collectively mean. For example, given the following information:

1. Paul will go to Mexico if no evidence that Sally will go to Mexico
2. Sally will go to Mexico if no evidence that Rob will go to Mexico.
3. Rob will go to Mexico if no evidence that Paul will go to Mexico.
4. Rob will go to Mexico if no evidence that Sally will go to Mexico.

then each rule by itself is easy to understand on its own, but it is difficult to tell the (stable) model of the program. For example, it is not easy to figure out who will go to Mexico from the statements above. The above rules can be modeled as the following logic program:

```
paul_go_to_mx :- not sally_go_to_mx.
sally_go_to_mx :- not rob_go_to_mx.
rob_go_to_mx :- not paul_go_to_mx.
rob_go_to_mx :- not sally_go_to_mx.
```

Under the stable model semantics of negation as failure [2, 1], we will conclude that Paul and Rob will go to Mexico and Sally will not.

The ASP paradigm greatly facilitates modeling of intelligent applications as solutions are specified not only by stating their positive aspects (e.g., relationships between various entities involved) but by stating negative aspects as well (e.g., relationships between entities that do *not* hold). Thus, for example, coloring relationship in the graph coloring problem (with 3 colors: red, green, yellow) between various nodes can be very declaratively specified by simply stating the following rules [1] (note that these rules are cyclical as well):

```
color(Node, green) :- not color(Node, red), not color(Node, yellow).
color(Node, red) :- not color(Node, green), not color(Node, yellow).
color(Node, yellow) :- not color(Node, green), not color(Node, red).
```

2 A Case for Predicate Answer Set Programming

Numerous systems have been built to execute answer set programs that are extremely sophisticated and efficient. CLASP is the best representative of these systems [10]. These systems restrict programs to predicates that only have variables and constants as arguments (compound terms are not allowed). Answer sets (or stable models) of such programs are computed by grounding the program rules with the (finite) Herbrand universe, suitably transforming it to an equivalent propositional theory, and then using a SAT solver to compute models of this theory. These models of the transformed program are the stable models of the original Answer Set Program. There are many problems with this model-finding approach that relies on a SAT solver:

1. Since SAT solvers can only handle propositional programs, these approaches only work for finitely-groundable programs. That is, programs with structures and lists occurring in arguments of predicates cannot be executed, as grounding of such programs will result in an infinite-sized program (due to the Herbrand universe being infinite). In many instances, lists and structures are essential for representing information.
2. Grounding of the program can lead to an exponential blowup in program size. For programs to be executable in such a system, a programmer has to be aware of how the grounding process works and how the ASP solver works and then they have to write their code in such a way that this blowup is minimized. This places undue burden on the programmer, as the programmer has to have knowledge of the grounding procedure as well as the model-finding process.
3. If the number of constants in the program is large, then a SAT-based approach is infeasible due to the size of the grounded program that will be created. It is next to impossible to build a general-purpose knowledge-based system using such an approach, as such a knowledge-based system will potentially have tens of thousands of constants.
4. SAT-based ASP solvers do not allow reasoning with real numbers.
5. SAT-based model-finding approaches compute the entire model. That is obviously an over kill. Most of the time users are interested in a specific piece of information. Thus, if we have a general purpose knowledge-based system, then the current ASP systems will compute the entire model, i.e., everything that can be inferred from the knowledge-base will be computed.
6. Often, it is hard to isolate the solution that is embedded in the model that is produced by the SAT solver. For example, if one solves the Tower of Hanoi problem using a SAT-based ASP solver, then the answer set will contain a large set of moves that are in the model. One cannot easily isolate the sequence of moves that represent the solution to the problem.
7. Since ASP systems compute the entire model, even a minor inconsistency in a narrow part of the knowledgebase will result in the system concluding that no answer set exists. A practical, large, real-world knowledgebase is very likely going to contain inconsistencies.

We believe that restricting ASP to predicates that can only contain constants and variables (no general terms) and using a model finding approach based on a SAT solver severely restricts the applicability of ASP to automated reasoning tasks and for building practical knowledge-based systems. We argue that designing query-driven, goal-directed ASP system that support general predicates that can contains general terms as arguments will result in a much more powerful, practical reasoning system. In fact, practical applications that involve common sense reasoning can be quickly built.

We have been working on designing query-driven answer set programming systems [3]. A query-driven system computes the partial answer set that contains the query (thus, it does not compute the entire answer set). Having a query-driven system addresses problems 5, 6 and 7 mentioned above [4], however, issues mentioned in points 1, 2 3, and 4 above still remain as problems. To alleviate problems 1, 2, 3 and 4 above, we have extended our system to allow general-purpose predicates. Thus, our extended system, called s(ASP), admits answer set programs containing predicates that are allowed to have variables, constants and structures as arguments [5, 6].

Our s(ASP) system does not ground the program. It can be thought of as full Prolog extended with negation-as-failure under the stable model semantics regime [6]. Problem 1, 2, 3, and 4 above are eliminated by s(ASP), since programs do not have to be grounded prior to execution. The s(ASP) system is publicly available [5], and has been used to develop a number of non-trivial applications based on ASP. Some of these applications cannot be executed on traditional ASP systems such as CLASP, as these applications make use of lists and structure to represent information. They have been developed by people who are not experts in ASP. These applications include:

- A system for automatically performing degree audit of a student's undergraduate transcript at a US University, i.e., automatically determining if a student can graduate with a degree or not. The system represents the graduation requirements laid out in the course catalog as ASP clauses. Use of negation is important for representing these requirements. The system has to make use of lists, and has hundreds of courses that appear as constants in the program (hence its grounding will produce an inordinately large program).
- A system for disease management, particularly, for chronic heart failure. This system automates the 80-page guidelines (that the American College of Cardiology has developed) by representing them in ASP. Our tests indicate that the system is able to make recommendations that doctors have missed [8].
- A system that represents high-school level knowledge about cells (in the discipline of biology) as answer set programs. It can answer high-school level questions posed as s(ASP) queries. The goal is to represent the knowledge in the entire introductory biology textbook as an answer set program, and then be able to automatically answer questions that would be asked of a student (the questions have to be translated into ASP queries that are then executed to find the answer).
- A recommendation system for birthday gifts: This system codes a human's knowledge about friends, level of friendship, a person's wealth level, generosity level, and hobbies as answer set programs. When queried, the system can recommend a birthday present for a particular friend.

Many other applications by non-expert users have been developed using the s(ASP) system.

3 Conclusions

We believe that the ASP paradigm is a very powerful paradigm that allows for complex human thought processes to be elegantly emulated [8]. Common sense reasoning is an important application of automated reasoning and we believe that being able to model common sense reasoning is critical to building practical automated reasoning systems. However, as argued above, the current model-finding, SAT-solver based approaches are not able to realize the full-power of ASP. We argued that query-driven implementations of predicate ASP are crucial to the ASP paradigm's success and that such implementations are needed to make the ASP technology practical for building large, scalable knowledge-based applications. An additional advantage of a query-driven approach over model-finding approaches is that in the latter case, everything has to be modeled in the ASP paradigm, while in the former case both the standard logic programming paradigm and the ASP paradigm can be made to work together.

ASP can be thought of as a better way of realizing "expert systems". The expert system enterprise failed in the past, we believe, due to the fact that standard logic-based formalisms that are monotonic in nature are inadequate for modelling human reasoning and common sense reasoning. Earlier implementations of expert system relied on these monotonic logics that are brittle and cannot tolerate inconsistencies (in the presence of an inconsistency every proposition as well as its negation becomes true). ASP, based on negation as failure, stable model semantics, and a non-monotonic logic, in contrast, is quite capable of faithfully modeling human reasoning. However, generalization to predicates and a query-driven execution model are critical to the success of using ASP for building expert systems, we believe.

Significant progress has been made in developing query-driven predicate ASP systems with the implementation of the s(ASP) system. However, considerable amount of research remains to be done. We urge the community to invest effort in developing query-driven predicate ASP systems. Immediate problems that can be tackled include extending s(ASP) with constraints over finite domains [9] and tabling [11].

Acknowledgment: Support from NSF Grant IIS 1423419 is gratefully acknowledged.

References

1. Baral, C. 2003. *Knowledge Representation - Reasoning & Declarative Problem Solving.* Cambridge Univ. Press.
2. Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In *International Conference and Symposium*, 1070–1080.
3. K. Marple, G. Gupta Goal-directed execution of answer set programs. Proc. PPDP 2012: 35-44
4. K. Marple, G. Gupta. Dynamic Consistency Checking in Goal-Directed Answer Set Programming. TPLP 14(4-5): 415-427 (2014)
5. K. Marple, E. Salazar, G. Gupta. The s(ASP) system. `https://sourceforge.net/projects/sasp-system/`
6. K. Marple, E. Salazar, G. Gupta. Computing Stable Models of Normal Logic Programs without Grounding. Forthcoming paper. March 2017.

7. G. Gupta, L. Simon, A. Mallya, R. Min, A. Bansal. Coinductive Logic Programming and Its Applications. Invited Tutorial. Proc. ICLP 2007. pp. 27-44.
8. Zhuo Chen, Kyle Marple, Elmer Salazar, Gopal Gupta, Lakshman Tamil. A Physician Advisory System for Chronic Heart Failure management based on knowledge patterns. TPLP 16(5-6):604-618, 2016.
9. K. Marriott, P. Stuckey. Programming with Constraints. MIT Press. 1998.
10. Martin Gebser, Benjamin Kaufmann, André Neumann, Torsten Schaub. clasp: A Conflict-Driven Answer Set Solver, LPNMR07, 2007. `https://potassco.org/`.
11. Terrance Swift, David Scott Warren. XSB: Extending Prolog with Tabled Logic Programming. TPLP 12(1-2): 157-187 (2012)