Künstliche Intelligenz manuscript No.
(will be inserted by the editor)

Interview with Vladimir Lifschitz

Vladimir Lifschitz · Torsten Schaub · Stefan Woltran

Received: date / Accepted: date

Abstract This interview with Vladimir Lifschitz was conducted by Torsten Schaub at the University of Texas at Austin in August 2017. The question set was compiled by Torsten Schaub and Stefan Woltran.

About

Vladimir Lifschitz is among the most influential scientists in the field of Artificial Intelligence (AI). He started out his work at Stanford with John McCarthy and made significant contributions in the area of Non-monotonic Reasoning (NMR). This can be seen a precursor to his later groundbreaking work connecting NMR with Logic Programming that led to the definition of the stable models semantics, and thus the initial foundations of Answer Set Programming (ASP).

Interview

First of all, thank you very much for taking your time for an interview for the special issue on Answer Set Programming of the German journal on Artificial Intelligence.

Thank you for coming to Austin to talk to me.

Torsten Schaub
University of Potsdam E-mail: torsten@cs.uni-potsdam.de

Stefan Woltran
TU Wien E-mail: woltran@dbai.tuwien.ac.at

How did you get started in Non-monotonic Reasoning, as the precursor of Answer Set Programming?

So I learned about non-monotonic reasoning in the early 1980s when John McCarthy just started working on circumscription. Before that I had known almost nothing about AI; it was not my area at all. My work in college and graduate school was on logic, proof theory, and intuitionistic logic. And then for a while I was interested in Operations Research and solving difficult, NP-hard search problems and I did some work on the investigation of algorithms, in particular, average run time of algorithms for solving difficult search problems. I worked at that time at the University of Texas at El Paso, where I worked in these areas. At some point, I learned that Michael Genesereth and John McCarthy were offering a week long summer school on AI in California and I decided to enroll just to learn a little bit about this area that was completely new to me. It was interesting. So they took turns and what I heard from Michael Genesereth was a standard nice introduction to AI. But what John McCarthy talked about was quite amazing for me because one half of it was an undergraduate introduction to logic, much too elementary for me, something that I learned many years before. But after that he started talking about ideas that were completely new to me, so that allowed me to see logic in a completely new way. One aspect of this is that he talked about applications of logic to formalizing commonsense reasoning. So he wrote situation calculus formulas on the board, for instance. That was completely new to me because I knew that you can apply logic to formalizing mathematics, and I knew that there is also philosophical logic, when you talk about knowledge and necessity, but I had never seen anything like the situation calculus. And then he started explaining circumscription. So he wrote the circumscription formula on the board and that was kind of a challenge to me because I thought that this is a syntactic transformation of formulas, whose properties — I thought, since I considered myself an expert in logic — should be obvious to me. So I thought to me, it was kind of a challenge, and I thought that I would think about this for a couple of days and all properties of circumscription will be completely clear to me, and it turned out that it took many years to figure it out. So that's how I started working on this.

For many years, it was just Non-monotonic Reasoning for me, and I did not think at all about the relation of this work to Logic Programming. Although I must say that when I decided that I wanted to work on circumscription, I moved to California from UT El Paso and started working at San José State University, which is geographically close to Stanford, so that I could come to McCarthy's seminars. My colleague there, Michael Beeson, said to me that if you are interested in Non-monotonic Reasoning, you should think about negation-as-failure. But I forgot about that comment for many years and later it turned out that this was a good idea — so that's how it all started.

Your initial focus in Non-monotonic Reasoning lay on circumscription. Why is the appealing idea of "minimal model reasoning" less present nowadays?

I would say that there were two reasons for this: First, Logic Programming is better as a knowledge representation formalism, when you want to, in particular, solve the frame problem, which generally was at that time McCarthy's main motivation for research on Non-monotonic Reasoning. McCarthy's first attempt to apply circumscription to solving the frame problem turned out to be completely incorrect. So there was this famous Yale Shooting domain invented by Steve Hanks and Drew McDermott that showed that the solution was incorrect. So many people including myself started thinking about ways to use circumscription, or other approaches using circumscription that would solve the frame problem correctly. And that was difficult. But when you formalize the common sense law of inertia using negation-as-failure, you do it in a very natural way and it works. So that was very impressive to me and many other people.

And the second advantage was that Logic Programming came with a ready implementation. So there was Prolog, and you could immediately experiment computationally with it, even before the invention of answer set solvers. Still, this other partial implementation was available, Prolog, and when you worked on circumscription everything had to be done manually: you had to verify all claims by mathematical computations on paper and being able to experiment that was a tremendous advantage.

What led Michael Gelfond and you to the idea of the stable models semantics?

So our work on the stable models semantics started with a discussion of Michael's paper published in 1987, the year before the stable models paper, which was called "On stratified auto-epistemic theories". And that was a very important paper because what Michael did is: he looked at the semantics of stratified logic programs proposed a year or two earlier by Apt, Blair, and Walker and he showed that you could define the same semantics in a much simpler way. The definition of what the intended model of a stratified logic program is was very complicated because you had to look at a particular stratification of your program. And you extracted this model by this iterated use of fix points corresponding to strata. And then there was this question — given that a program can have many different stratifications — so is it true that for all of them we get the same intended model? So the model is really determined by the program itself and not by stratification. And it is true but that was not easy to prove. So it was rather complicated. So what Michael said: If you have a rule with negation, just think of negation-as-failure as an abbreviation for the composition of two operators: Classical negation and then the modal operator L used in Auto-epistemic Logic, invented by Bob Moore a few years earlier. So you have this very simple syntactic translation from Logic Programming to formulas in Auto-epistemic Logic, and then apply Moore's semantics. So that was very nice, very impressive. And also we saw at some point that there were some non-stratified programs to which of course the definition by Apt, Blair, and Walker was not applicable but this construction by reduction to Auto-epistemic Logic worked and would give the model that corresponded to its intended meaning. And that could be processed by Prolog and Prolog would give correct answers. So we saw immediately the advantages of this. But the problem with Michael's work was that to understand his very simple definition you had to know Auto-epistemic Logic. And what occurred to us is that maybe we can reformulate the semantics, so that it would not include explicit reference to Auto-epistemic Logic. So we took Moore's definition of the semantics of Auto-epistemic Logic and looked at what happens if we applied it to this specific kind of formulas that correspond to rules in logic program and reformulated Moore's definition directly in terms of logic programs, so that you would not have to do this two-step process, translating and then using a very general definition that applies to arbitrary propositional formulas with a model operator. And that was the stable models semantics.

Let me add one other thing that is not well known. Actually the philosophical logician Kit Fine came up with the same idea independently around the same time. So actually his name is not often mentioned in the literature on the history of this subject because Kit Fine presented his work to philosophers rather than to logic programmers but he should be viewed as a co-inventor of stable models semantics.

Since we talk about this, we should also mention the work of Bidoit and Froidevaux . . .

That's right. So that paper actually was published also in '87 simultaneous with Michael's and that it was very similar to Michael's approach, instead of translating logic programming rules into formulas with the modal operator using Auto-epistemic Logic, they turned rules into defaults in the sense of Reiter and applied Reiter's semantics. And although Auto-epistemic Logic and Reiter's Default Logic are not equivalent at all — their relationship is rather complicated — but in that particular case, if you simply translate standard programs with negation-as-failure, the result is the same. So their work like Michael's essentially defined stable models but in an indirect way. And what happened later, with the paper on stable models and Kit Fine's paper, is that the right definition became available that did not refer to more complicated non-monotonic formalisms.

When and how did you realize the virtue of Answer Set Programming as a proper paradigm, as opposed to the initial idea of a semantics for logic programming?

Actually I can give you a precise date: It was on February 6, 1998. Because that was the date when I sent an identical message to Ilkka Niemelä and to Mirek Truszczyński with the same question. I knew that they were implementing stable models semantics in various ways and what I wrote to both of them was this, I said: I am looking for an implemented system that can find a stable model of a program, which has variables but no function symbols, is non-disjunctive but includes *[integrity]* constraints. And I asked Ilkka: Can your *[system]* Smodels do this? Is there anything else that we can try? And also I asked Mirek about this. That was of course the time when Smodels existed but its input language was the language of Prolog. It was the first version of Smodels, so there were no constraints, no rules with the empty head. But Ilkka explained to me how you can model constraints using auxiliary atoms for falsity. So his answer was "yes". And as I remember, the reason why I asked this question was that I wanted to use systems like Smodels for planning. I understood that I could describe possible plans of a given length by a program with many stable models — of course, there were no choice rules at that point, so that was supposed to be a very non-stratified logic program with many stable models — plus *[integrity]* constraints. So that was obviously the time when we started doing experiments like this. And to put this in perspective, this was as I said in 1998, so that was before the publication of two papers, one by Ilkka and the other by Marek and Truszczyński with the words "stable" and "paradigm" in their title. On the other hand, that was a year after the paper by Dimopoulos, Nebel, and Köhler called "Encoding Planning Problems in Non-monotonic Logic Programs" where they actually used Smodels for planning. But I was not aware of this, so this idea came to me independently a year later.

At the International Conference on Logic Programming (ICLP) in 2016, Ilkka Niemelä and Mirek Truszczyński, gave you credit for coining the term "Answer Set Programming". What was your motivation for this?

Yes. That was — as I remember — motivated by the use of the term "stable logic programming" in a paper by Marek and Truszczyński. And I thought that "answer set programming" is preferable because at that time I thought that "answer sets" is a better expression than "stable models". And the reason was that at that time I didn't think about rules as being similar to formulas. To me, rules were like Reiter's defaults, rather a generalization of inference rules. When you say "a stable model", it sounds like there are arbitrarily models and some of them are stable. And since the idea of model as in classical logic does not apply to sets of inference rules, I thought that that was not appropriate. So that's why

I supported the idea of using "answer set programming" rather than "stable logic programming". But I must say that now my view is different because now I know how useful the idea of treating rules in a logic program as abbreviations for formulas is and stable models are really models in the sense of classical logic with some special property. So now I use the term "stable model" a lot more often than "answer set". That was different at that time.

Why has Answer Set Programming not reached the same status in the US as in Europe?

I don't know. It's unfortunate.

Answer Set Programming is often considered as the continuation of non-monotonic logics. How do you see this evolution in the light of what happened in general AI?

This non-monotonic reasoning that we talked about in the beginning, the precursor of ASP, it was a major contribution to logic but it was invented by prominent members of the AI community, John McCarthy, Ray Reiter, and Drew McDermott. And it was described in this special issue of the AI Journal on Non-monotonic Reasoning in 1980 and it was in response to AI problems such as the frame problem. So at that time and in the early days of ASP, this work was seen as just a part of AI research. Later with the emergence of answer set solvers, it turned out that there is a close relationship between ASP and essentially Operations Research, difficult optimization and search problems. What happened is that this relationship between ASP and AI is gradually becoming weaker and weaker. More and more often, we see applications, where we think of ASP as a special brand of search methods, a declarative approach to search. That's how often ASP is described now, it's a declarative approach to search. And, on the one hand, of course, this is good and in many cases this is how ASP is useful because it is a powerful declarative approach to search. On the other hand, I think it is unfortunate and I am talking not just about the history of the subject but also my own work, when I compare what I was working on over the recent years and the kind of papers that I wrote early. I see this thing that my work is less and less work on AI and more and more work on search methods and applications to search. And I think it is unfortunate that you do not see very often now papers that would stress the role of ASP as an approach to the main AI problem, the problem of creating computational models of intelligence. So, I looked at the website of Michael Gelfond's group. Michael is one of the people who are trying to keep this relationship to general AI concerns alive. And the words that we see in the titles of the papers there reflect general AI concerns. The titles of the papers on the website of their group mention intentions of agents, and multi agent systems, and question answering, and inference of information from natural language conversations, and things of that kind. But even there, I think you see this kind of titles when you look at what happened fifteen and twenty years ago rather than about what is happening now in the work of other researchers. You see this even less often and I think that's unfortunate. It would be great if we could see this relationship between ASP and general AI concerns coming back and being again influential.

How come that the relation of Answer Set Programming to the database world, in particular, to Datalog, is often overlooked or neglected?

Yes, one aspect of this is that from the deductive database perspective, there is a crucial relationship of course between extensional predicates defined by a relational database and intentional predicates defined by rules. And this relationship, formally, I think it just does not exist. When you look at user guides on answer set solvers, it is not often emphasized. When you apply answer set solvers to solving problems, you do actually often have a separate file with an input, say the description of a specific graph, which corresponds to a relational database with a set of facts, and separately a program itself, which is the definition of the intentional predicates. But unfortunately this is somehow not emphasized very much in user guides. You could have input and a particular problem in the same file and run the solver and get the same results. So I think, it's important to remember this relationship.

In this volume, we also have a paper on the logical foundations of Answer Set Programming. Why did it take quite a while until the importance of corresponding monotonic logics, viz. the logic of here-and-there, was recognized by the NMR and ASP community, for instance, to test strong equivalence?

I don't know why it took so long but if you are interested I can tell you the story about how strong equivalence and the relevance of the logic of here-and-there was invented.

Yes, with pleasure!

It happened at lunch at the *[International Conference on Logic Programming and Non-monotonic Reasoning]* LPNMR in 1999 where Esra Erdem, who was my grad student at that time, and I gave a paper called "Transformations of Logic Programs Related to Causality and Planning". So, what we were doing there is we were looking at some rules occurring in logic programs related to describing dynamic domains and planning, where we saw that sometimes you can modify a rule, and the stable models will not change if the rule is a little different. So we proved in this paper two theorems of this kind that explained that actually modifying rules somewhat did not affect the answer sets of the program. So what we verified is what is now called "strong equivalence". But this term did not exist at this time and we proved directly based on the definition of answer set. And on the day when Esra was supposed to give a talk, we had lunch, she, David Pearce, and I, and we explained to David what we were proving and he said that there is also another approach to proving this. And he explained how to do this essentially by showing that the two programs are equivalent in the logic of here-and-there. And from his earlier work, it was clear that if two programs are equivalent in the logic of here-and-there, then no matter what you add to them, if you add the same code, the stable models will be the same. We liked this very much and in her talk two hours later, Esra mentioned that she just learned about another way of proving this. And then, the next step was to find out whether this method was always applicable. That is, whether or not it is true that whenever two programs are strongly equivalent — and the term by the way was suggested by Michael Gelfond — then this can be always proved by Pearce's method. That is, by proving that they are equivalent in the logic of here-and-there. And that was more difficult. And we started collaborating with Agustín Valverde on that and proved together this difficult part of the theorem to have the if-and-only-if characterization of strong equivalence. And that's how this thing was invented. Why it happened in 1999 and not earlier I don't know.

Actually, in this logic of here-and-there, negation is defined in terms of implication, which plays the central role. On the other hand, Answer Set Programming is usually conceived as being centered upon negation-as-failure. What is your view on this putative discrepancy?

Yeah, it is an interesting fact. In classical and even in intuitionistic logic, you can think of negation as an abbreviation, that is, the negation of F *[*$\neg F$*]* is F implies falsity *[*$F \to \bot$*]*. So negation is even intuitionistically a special case of implication. But somehow in earlier work on stable models, implication, which you find between the head of a rule and the body of a rule, and negation, which is applied to atoms in the body, were treated in completely different ways. And it changed when we wanted to try to represent rules with aggregates as formulas. Because to do that, for some aggregates, aggregates that are neither monotone nor anti-monotone, you have to use implications in the body of a rule, as Paolo Ferraris suggested. So at that point, it became clear that it's good to define stable models for arbitrary propositional formulas, including nested implications, and then there is nothing special about negation: negation can be used as an abbreviation. And that led to this view of stable models that you can find originally in Pearce's definition of a stable model in terms of equilibrium logic and then later work by Ferraris where he defined reducts in a new way, in which negation does not play a special role. And it turned out that if you apply this definition to the special case of traditional Prolog-like rules, you get the same concept of a stable model although now there is nothing special about negation for you. And that's to me an amazing fact, a kind of mathematical curiosity, very very strange and something you'll have to live with.

As far as I know, you did your dissertation on intuitionistic logic? So is Answer Set Programming taking you back to your roots?

Actually, to some degree yes. And that was actually meant to be part of my answer to your other question about major surprises while working on ASP. There were two major surprises for me, one is that some problems that I had worked on many years earlier turned out to be related to the work on ASP. As I mentioned to you, my very early work was related to intuitionism and then at some point I worked on hard NP search problems. And then it turned out that properties and applications of stable models, on the one hand, are closely related to intuitionism, they are described by this intermediate logic of here-and-there, which is intermediate between intuitionistic and classical logic, and, on the other hand, it became a tool for solving exactly the kind of knapsack problems and integer programming problems that I was interested in. So I would say that these were two major surprises to me. It was like meeting with old friends many years after I saw them and it turned out that these guys are again relevant.

Looking backward, what do you consider as the major highlights in the short history of Answer Set Programming?

I would like to talk about one event that was particularly striking to me. That this work, at least for me, why I started working on stable models, was not meant to be useful in any practical sense because my interest was related to the desire to understand better negation-as-failure, not in purely computational terms. And then it turned out to be useful. And this is something that happens sometimes in history of science and every time I find it remarkable. You know, sometimes your research is motivated by the desire to understand something more clearly and sometimes by the desire to create something useful. And in good science, what's particular nice, I think, is when your work is motivated by the desire to understand something but it happens to be also useful, as a kind of a side-effect of that. And the kind of prototypical example of that in history of science was the work by Isaac Newton whose goal — as far as I understand — was to find out what God was thinking when he was creating the universe. And to answer this question, his answer was that he was thinking of second order differential equations. And to arrive at this answer, he had to invent calculus and classical mechanics, which happened to be useful. And on a much smaller scale, that happens in science all the time, and every time it's wonderful and amazing that you just want to clarify something and it unexpectedly turns out to be useful. And in the history of Answer Set Programming, it happened also. And the reason why I think this could happen in this particular case was that we were very fortunate with people who decided to take part in the project. Many of them are excellent programmers and software designers and at the same time they are good mathematicians. Not strong mathematicians in the sense that they are interested in proving difficult theorems but they have good mathematical taste; they understand what mathematical clarity and elegance is. And the fact that we have several people in our community like this, that is, I think, the reason why this miracle happened.

Where has Answer Set Programming left its footprints and where could it have its major impact in the future?

I think the main accomplishments of our work is demonstrating the usefulness of a declarative approach to search and it seems to me that this is where ASP is different from SAT *[Satisfiability Testing]*. In some ways, SAT and work on satisfiability solvers and work on answer set solvers, they are similar because both of them do search declaratively but in Answer Set Programming you really deal with a program written in a declarative language. And when you use SAT solvers, in many cases, you don't have such a declarative program that you can look at. You have this huge set of clauses which is generated in a non-declarative way. So this declarative object, a set of clauses, is hidden; it is not the language in which the programmer writes. So using SAT solvers often, even though sets of clauses are declarative objects, you cannot think of that as declarative programming. And here we deal with a declarative approach to search and that is an important accomplishment.

So what I want to see in the future, one thing that I mentioned before, I would like to see the connections between ASP and general AI concerns reestablished. They played an important role earlier and I hope that will happen. And another thing that I hope will happen in the future is that Answer Set Programming will contribute to the further development of formal methods. When we want to verify formally the correctness of a program written in any language, you have to have a complete formal description of the semantics of the language. And it seems to me that, because Answer Set Programming is fully declarative, having a completely precise, mathematically precise definition of the semantics of input languages of answer set solvers is more realistic. It should be easier to do that than if we deal with procedural languages or languages not fully declarative such as Prolog or, for instance, functional languages with non-declarative elements. And for this reason, it seems to me that when we apply formal methods to verifying answer set programs, which is the main topic that is of interest to me now, I hope that we will be maybe more successful than with other programming languages because the language, I hope, will lend itself more easily to a completely formal view.

Looking forward, what do you consider the major milestones in the future?

First, merging answer set solving systems with other programming tools. They are very useful, often not in isolation but as elements of systems which have also some non-declarative parts in them. Secondly, improving the methodology of Answer Set Programming. The work on methodology of debugging answer set programs is still in an infant stage and I hope that there will be some new accomplishments there. Another important difficult problem is when you wrote a program that is correct and you want to optimize this. How do you do this? So knowledgeable people know which flag you should set to cause the solver or the grounder to use strategies that are particularly good for your program, they know what to experiment with, and there is some kind of wisdom people may have about what is a good way to write a rule, what is a bad way to write a rule. But ideally, our answer set solvers should be like optimizing compilers. Even if you wrote your program in a non-efficient way, all the optimization should happen inside. So that you could only, when you are writing a program, think about its clarity and correctness, not about optimization. The solver should take care of this. I hope the day will come when we will be free to think in terms of correctness, and the solver will do optimizing for us.

Thank you so much for this interview! I actually learned quite a lot from it!

Vladimir Lifschitz is a professor of computer science at the University of Texas at Austin. His research interests are in computational logic and knowledge representation. He is a member of Heterodox Academy and leads Texas Action Group at Austin.