Intelligenza Artificiale 5 (2011) 77–81
IOS Press

Evolving Reactive Logic Programs

J.J. Alferes [a], F. Banti [a], A. Brogi [b]
[a] *CENTRIA, Universidade Nova de Lisboa, Portugal*
[b] *Dept. of Computer Science, University of Pisa, Italy*

Abstract. In this paper we briefly describe the research activity that we have been carrying out during the last years on dynamic logic programs. After reviewing our contributions to strengthening the semantic foundations of dynamic logic programs, we describe a simple formalism to reason about actions —based on dynamic logic programs— and its event-condition-action extension that supports the specification and the execution of reactive programs.

Keywords: Logic programs, dynamic knowledge, action description languages, event-condition-action languages.

1. Introduction

Research in Artificial Intelligence (AI) is concerned with producing machines to automate tasks requiring intelligent behavior. An important problem to face when implementing AI applications is how to represent knowledge, and how to extract information from such knowledge. This area of research is known as knowledge representation (KR) and reasoning. The dominant approach in KR is to define symbolic paradigms based on some form of logic, usually consisting of crude facts and more sophisticated logic formulas. Together, facts and formulas form the knowledge base (KB) of the AI application. Many tasks for AI applications also demand to perform some kind of actions. Hence actions, and possibly the effects of actions, should be represent able in the KR framework, and the mechanism specifying when an action must be performed must be defined. Furthermore, usually interactive applications continually receive external inputs in the form of messages, perceptions, commands and so on. Such inputs can be considered as events to which the AI application is supposed to react in an intelligent way. Reactivity is a key feature in dynamic domains, where changes frequently occur. Among the existing proposals for programming reactive behavior, *Event-Condition-Action* (ECA) languages distinguish themselves for their flexibility and intuitive syntax and semantics.

Dynamic domains require AI applications capable to handle frequent changes and, consequently, to update their KBs. The required updates surely involve the extensional part of the knowledge base (facts), but occasionally it may be necessary to update also the intensional part (logic formulas) to represent the fact that the very rules of the domain changed. Furthermore, for adapting to the new situation, besides knowledge updates, it may be necessary to update the behavior of the AI applications, i.e., the reactive mechanisms themselves. These updates may be the result of external inputs, but it might be necessary for the application to perform actions leading to self-updates. Moreover, besides what could be called basic actions like, for instance, insertions and deletions of facts and formulas, developers may want to specify more sophisticated actions obtained by combining the basic ones.

Among the existing formalisms for KR, Logic Programming (LP) has a simple logic-based syntax, formal declarative semantics and implemented inference systems. In the past years, part of the research on LP focused on representing dynamic knowledge, i.e., knowledge that is constantly self-updated, leading to the *dynamic logic programming* (DyLP) framework [5,9,10,12]. Taking advantage of the established results in the field, we developed a (dynamic) LP framework for programming AI applications satisfying the above listed features.

In this paper, we first review (Section 2) our contributions to strengthening the semantics foundations of dynamic logic programs that yielded a refined stable-model based semantics and a well-founded semantics for this class of logic programs. We then describe (Section 3) a simple formalism, named EAPs (after Evolving Action Programs) to reason about the effects of actions —based on dynamic logic programs and on the LP update language *Evolp* [4]— and its event-condition-action extension *ERA* (after Evolving Alge-

braic Programs) that supports the specification and the execution of reactive programs. As we will see, *ERA* supports the specification and the execution of reactive programs by detecting (simple and complex) events and by executing (simple and complex) actions including self-updates. Since ERA can also encode EAPs, it hence satisfies the features listed earlier in this Introduction. Finally some conclusions and directions for future work are discussed (Section 4).

We assume the reader is familiar with logic programming and the stable models and well-founded semantics and refer to [6] for details on syntax and semantics of LPs.

2. Dynamic Logic Programs

Dynamic Logic Programs represent evolving knowledge. Syntactically, a DyLP $\mathcal{P}$ is a *sequence*

$$P_1, \ldots, P_n$$

(rather than a single program) of generalized logic programs (GLPs), viz., programs where rule heads may be negative literals. P_1 represents the initial knowledge and the other $P_i s$ are supervenient updates representing the evolution of the described situation. Given two updates P_i, P_j, of a DyLP $\mathcal{P}$, P_j is said to be *more recent* than P_i if P_j follows P_i in the sequence $\mathcal{P}$. In the past years, several semantics have been defined for providing a meaning to DyLPs [5,9,10,12,13]. These semantics are extensions of the stable model semantics of normal logic programs, in the sense that, whenever the considered DyLP is a single normal program P, the models of P in the considered semantics for DyLPs coincide with the stable models of P. Another common denominator of these semantics, is the *causal rejection principle* [10,12]. This principle states that *a model M of a DyLP $\mathcal{P}$ must fulfill a rule τ in an update of $\mathcal{P}$* unless *there exists a rule in a more recent update that is in conflict with τ and whose body is true in M*. Two rules τ and η are said to be *in conflict* if they have complementary heads, viz., the head of τ is a literal A and the head of η is $not\, A$ or viceversa. The principle allows a more recent rule to specify an *exception* to an older one, thus allowing to update previous beliefs.

The semantics for DyLPs based on the causal rejection principle coincide on large classes of programs but disagree on some examples and, at the time we started our investigation, there was no general agreement on which should be *the* stable model-like semantics for DyLPs based on the causal rejection principle. Moreover, all the semantics defined before we started our investigation show *counterintuitive behavior* in some well known example. The simpler examples involve tautological updates that happen to change the semantics of a DyLP, while immunity to tautologies is a property generally required to a semantics.

For instance, the single program DyLP

$$\begin{aligned} P_1 : \; & not\, rain. \\ & rain \leftarrow cloudy. \\ & cloudy \leftarrow not\, sun. \\ & sun \leftarrow not\, cloudy. \end{aligned}$$

has one model $\{not\, rain, sun\}$. If we update P_1 with

$$P_2 : \;\; rain \leftarrow rain.$$

another model $\{rain, not\, sun\}$ is allowed. Somehow, the tautology has generated another model by rejecting the rule $not\, rain$. In general, all the known counterintuitive behavior occur in DyLPs with cyclic dependencies among literals, somehow leading to the introduction of undesired models, although a formal definition of *counterintuitive behavior* and *undesired model* was missing. Our contribution was:

- to formalize the concept underlying such counterintuitive behavior and to clarify which should be the right semantics for DyLPs by establishing which properties should be satisfied by such semantics, and
- to define a semantics satisfying those properties, thus avoiding the known counterintuitive behavior.

To achieve these results we defined the *refined extension principle* [1]. The refined extension principle is a criterion stating when the addition of rules to a program should not add more models to its semantics, and it permits to formalize which models should not be introduced. Then, we defined the *refined stable model semantics* (or simply refined semantics) for DyLPs that refines the other stable-like semantics for DyLPs. Formally this was achieved by associating to each DyLP $\mathcal{P} = P_1, \ldots, P_n$, an operator over sets of literals $\Gamma^R_{\mathcal{P}}$ and defining the refined models of $\mathcal{P}$ as the fix-points of $\Gamma^R_{\mathcal{P}}$. The $\Gamma^R_{\mathcal{P}}$ operator is formally defined as follows:

$$\Gamma^R_{\mathcal{P}}(M) = \text{least}\left(\rho(\mathcal{P}) \setminus Rej^R(\mathcal{P}, M) \cup Def(\mathcal{P}, M)\right)$$

where $\rho(\mathcal{P})$ is the multiset of all the rules appearing in any program of the sequence $\mathcal{P}$ and $Rej^R(\mathcal{P}, M)$

is the multiset of all the rules τ in some update P_i of $\mathcal{P}$ for which there exist a rule η in some update P_j with $i \leq j$ such that τ and η are in conflict and the body of η is true in M. Finally $Def(\mathcal{P}, M)$ is the set of default assumptions, i.e., the set of all the negative literals $not\ A$ such that there exists no rule in $\mathcal{P}$ whose head is A and whose body is true in M.

The refined semantics was proved to satisfy the refined extension principle and the causal rejection one. Furthermore, we extended the concept of *well supported models* [6] to DyLPs and proved that the refined models of a DyLP are exactly its well supported models.

A further result was the definition of a *well founded semantics for DyLPs* [8]. The well founded semantics is a skeptical approximation of the stable model one. From a practical point of view, the well founded semantics has less expressivity (for instance it does not allow to express logic constraints) and less inference power (it allows to derive less conclusions). On the other hand, the well founded semantics is computationally less expensive than the stable model semantics. Determining a (refined) stable model of a (dynamic or generalized) logic program is an NP-complete problem, while the computation of the well founded model of a normal logic program has polynomial complexity.

Moreover, unlike the stable model one, the well founded semantics is always defined and, according to it, a program can be queried about specific information without having to compute its whole semantics. Due to these features, the well founded semantics is a better candidate than the stable model one for applications that are time-committed and require to process huge amounts of data, like most real world database related applications.

We defined a well founded semantics for DyLPs that extends the well founded semantics for normal LPs and approximates the refined one, in the sense that (as for normal LPs) the well founded model of a DyLP is a subset of any of its refined models. Moreover, the well founded semantics for DyLPs preserves the good features shown for the class of normal and generalized LPs, i.e., the well founded model always exists, its computation is polynomial, and a DyLP can be queried about specific information without having to compute its whole semantics.

The well founded model was defined as the least fixpoint of an operator $\Gamma\Gamma^R$, combining the Γ^R operator used for defining the refined model semantics with another operator Γ used for defining another semantics for DyLPs, i.e., the *dynamic stable model semantics* [12].

3. Reasoning about and executing actions

After strengthening the formal foundation of dynamic logic programs, we turned our attention to the problem of programming self-updatable AI applications capable of reasoning about and executing actions. A bridge, using DyLPs, between dynamic KR and this kind of applications was already established by the family of *LP updates languages* [4,10,12]. These languages are built on the top of a DyLP semantics and, besides representing dynamic and constantly updated knowledge, they allow one to specify how a KB should be updated. Among these formalism, the Evolp language [4] has a particularly simple, but highly expressive, syntax and semantics, and hence it was chosen as the starting point of our investigation. Evolp is a language for building sequences of DyLPs starting from an original program. Syntactically, Evolp extends the language of LP with new atoms $assert(r)$ where r is a rule. An Evolp programs evolves passing from the current state to the next one by updating the program with all the rules r such that the atom $assert(r)$ is true in the current state.

A widely used way to describe and reason about the effects of actions are *action description programs* written in specific formalisms called *action description languages* [11]. We defined an action description language of our own, christened Evolving Action Programs (EAPs) [2]. EAPs are defined as a macro language on top of Evolp in the sense that every statement in EAPs is a syntactic notation for a set of Evolp statements and the semantics of an EAP is given by the semantics of the corresponding Evolp program.

Syntactically, an EAP statement can be:

- an inertial declaration $inertial(f)$, or
- a static, LP-like rule $L \leftarrow L_1, \ldots L_n$, or
- a dynamic rule **effect(**$H \leftarrow B$**)** $\leftarrow Cond$.

The meaning of an inertial declaration $inertial(f)$, where f is an atom (usually called a *fluent* in the context of action description languages) is that the truth value of f is preserved in time unless it changes as an effect of the execution of an action. A static rule describes the (static) rules of the environment by expressing correlations among fluents. A dynamic rule expresses the effect of the execution of actions. Syntactically, the effect $H \leftarrow B$ is a static rule, while $Cond$

is a conjunction of action literals representing actions being or not executed and fluent literals representing preconditions for the considered effects to take place.

The expressivity of EAPs was compared with that of the action languages $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ (see [11] for a detailed description of these languages) and for each of these languages, a modular translation of their action programs into EAPs was defined, thus proving that EAPs are *at least as expressive* as these languages. Moreover, EAPs show a novel capability of encoding, by sequences of EAPs, successive elaborations or updates of an action description problem, similarly to the capability of DyLPs of expressing successive updates of a logic program.

Besides reasoning about the effects of actions, we also needed a formalism for executing them. This was achieved by defining an ECA formalism called ERA (after Evolving Reactive Algebraic programs) [3]. Along with inference logic programming rules, ERA presents two new types of rules for specifying the execution of actions, i.e., *active* and *inhibition* rules of the form, respectively:

$$\textbf{On } Event \ \textbf{If } Condition \ \textbf{Do } Action. \quad (1)$$

$$\textbf{When } B \ \textbf{Do } not\ Action. \quad (2)$$

where $Event$ is an *event literal* encoding the occurrence of an event and $Condition$ is a conjunction of literals expressing the condition under which an $Action$ (syntactically an atom) is executed. Finally, B is a conjunction of literals expressing conditions under which $Action$ should *not* be executed. Both events and actions can be basic or complex ones. Complex events and actions are obtained by combining simple ones via an *event* and an *action* algebra.

Events occur at a given instant and are volatile information. Basic events may be external, representing incoming inputs and commands, or internal, raised by the system itself. The event algebra allows to combine events occurring simultaneously or at different time points. For instance, the complex event $A(e_1, e_2, e_3)$, where $A/3$ is a ternary operator and the $e_i s$ are events, occurs at instant t iff e_3 occurs at instant t, e_1 occurred at some previous instant and e_2 did not occur in between.

Actions represent operations to be executed. Basic action can be external, representing some external operation to be executed, or internal, specifying the occurrence of events or self-updates. As for events, basic actions can be combined by an algebra of operators, thus obtaining complex actions specifying a flow of operations. For instance, given two actions a_1 and a_2, action $a_1 \triangleright a_2$ specifies that action a_2 must be executed after a_1, while action $a_1 \| a_2$ specifies that a_1 and a_2 can be executed concurrently.

Among internal actions, particularly important ones are the *assertion* and the *deletion* of facts and rules. While deletion removes facts and rules from the KB, the assertion of rules causes the application to update itself by a new fact, an inference, an active or an inhibition rule. New facts and inference rules are incorporated by the underlying DyLP semantics (that can be the refined as well as the well founded one). Also new active and inhibition rules are incorporated by the underlying DyLP semantics. Assertions of rules of the forms (1) and (2) are translated, respectively, into the LP updates

$$Action \leftarrow Condition, Event.$$
$$not\ Action \leftarrow B.$$

The underlying DyLP framework allows to establish whether the atom $Action$ is derived or not. In the former case, the corresponding action is executed. Unlike other ECA languages, an ERA program can update not only its KB but also its behavior by asserting new active rules and specifying exceptions to existing active rules by asserting inhibition ones. It was also proved that every Evolp program, and hence every EAP, can be directly encoded into ERA. Thus ERA is a paradigm capable of both executing and reasoning about actions. In [7] ERA is discussed in detail and compared to existing formalisms for programming reactive behavior. We simply point out here the two main novelties of ERA, i.e., its self evolution capabilities and the featured possibility of both programming the execution of actions and reasoning about their effects.

4. Conclusions and future work

In this paper we have briefly described the research activity that we have been carrying out during the latest years on dynamic logic programs. After reviewing our contributions to strengthening the semantics foundations of dynamic logic programs, we have presented the EAPs formalism to reason about actions, and its recent event-condition-action extension *ERA* that supports the specification and the execution of reactive programs. While space limitations only allowed us to provide an extended abstract of this research activity, more details can be found in the papers [1,2,3,8] and

a complete presentation of all the results is reported in [7].

There are several branches of research for future work. One of them is the definition of *action query languages* [11], that is, languages for extracting information about the possible evolution of the situations described by EAPs and to address planning issues, e.g., how to determine, given a current state and a goal, a sequence of actions leading to a state satisfying that goal. Another direction for future work are *transactions*. Transactions are a fundamental issue in applications requiring databases updates and resources allocation. Although the action algebra of ERA has enough expressivity to program complex actions, it is still less than adequate for defining either ACID transactions or transactions with associated compensation activities.

References

[1] J. J. Alferes, F. Banti, A. Brogi, and J. A. Leite. The refined extension principle for semantics of dynamic logic programming. *Studia Logica*, 79(1):7-32, 2005.

[2] J.J. Alferes, F. Banti, A. Brogi. From logic programs updates to action description updates. In J. Leite, P. Torroni (eds.), *CLIMA V*, LNAI 3487, pages 52–77, 2005.

[3] J.J. Alferes, F. Banti, A. Brogi. An event-condition-action logic programming language. *JELIA 2006*, LNAI 4160, pages 29-42, 2006.

[4] J. J. Alferes, A. Brogi, J. A. Leite, L. M. Pereira. Evolving logic programs. *JELIA'02*, LNCS 2424, 2002.

[5] J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. C. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *The Journal of Logic Programming*, 45(1–3):43–70, 2000.

[6] K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *The Journal of Logic Programming*, 19 & 20:9–72, May 1994.

[7] F. Banti. *Evolving Reactive Logic Programs*. PhD thesis, Universitade Nova de Lisboa, 2008.

[8] F. Banti, J.J. Alferes, A. Brogi. Well founded semantics for logic program updates. *IBERAMIA'04*, LNCS 3314, pages 397–407, 2004.

[9] F. Buccafurri, W. Faber, and N. Leone. Disjunctive logic programs with inheritance. *ICLP'99*, 1999.

[10] T. Eiter et al.. A framework for declarative update specifications in logic programs. In *IJCAI*, 2001.

[11] M. Gelfond and V. Lifschitz. Action languages. *Electronic Transactions on AI*, 16, 1998.

[12] J. A. Leite. *Evolving Knowledge Bases*. Frontiers in Artificial Intelligence and Applications, vol. 81, 2003.

[13] J. A. Leite and L. M. Pereira. Generalizing updates: from models to programs. *LPKR'97*, 1997.