MODELLING ADAPTIVE CONTROLLERS WITH EVOLVING LOGIC PROGRAMS

Pierangelo Dell'Acqua, Anna Lombardi
Department of Science and Technology (ITN) - Linköping University
601 74 Norrköping, Sweden

Luís Moniz Pereira
Centro de Inteligência Artificial (CENTRIA) - Departamento de Informática, Universidade Nova de Lisboa
2829-516 Caparica, Portugal

Keywords: Adaptive controllers, non-monotonic reasoning, evolving logic programming.

Abstract: The paper presents the use of Evolving Logic Programming to model adaptive controllers. The advantage of using well-defined, self-evolving logic-based controllers is that it is possible to model dynamic environments, and to formally prove systems' requirements.

1 INTRODUCTION

Intuitively, an adaptive controller can change its behaviour in response to changes in the dynamics of the process and the disturbances (Åstrom and Wittenmark, 1990).

One of the first approaches proposed for adaptive control is gain scheduling, which implements an open-loop compensation. Other approaches have been proposed based on model-reference adaptive system (MRAS) or self-tuning regulator (STR). They both can be seen as composed of two loops. These classical approaches to deterministic adaptive control have some limitations when unknown parameters enter the process model in complicated ways. In this case, it may be difficult to construct a continuously parameterized family of candidate controllers.

An alternative approach to control uncertain systems has been proposed (Hespanha et al., 2003). The main feature which distinguishes it from conventional adaptive control is that controller selection is carried out by means of logic-based switching rather than continuous tuning. Switching among candidate controllers is performed by a high-level decision maker called a supervisor, hence the name supervisory control. The supervisor updates controller parameters when a new estimate of the process parameters becomes available, similarly to the adaptive control paradigm, but these events occur at discrete instants of time. This results in a hybrid closed-loop system.

Another class of adaptive controllers is fuzzy logic controllers (Mamdani and Baaklini, 1975). They are based on adaptation algorithm to update parameters of the controller as classical adaptive control but they are capable of incorporating linguistic information from human operators or experts. This characteristic is particular important for systems with a high degree of uncertainty, i.e. systems that are difficult to control from a control theoretical point of view but they are often successfully controlled by human operator.

Evolving Logic Programming (EVOLP) is an extension of logic programming (Alferes et al., 2002). It allows one to model the dynamics of knowledge bases expressed by programs, as well as specifications that dynamically change. In this paper EVOLP is used to model adaptive controllers. A case study will be illustrated where the controller is implemented by using EVOLP.

The paper is structured as follows: Section 2 introduces the notion of adaptive control with particular focus on model reference adaptive control and adaptive fuzzy control, Section 3 presents the language and semantics of Evolving Logic Programming. Section 4 describes a case study to model adaptive control systems using evolving language programming. Finally, Section 5 discusses some future work.

2 ADAPTIVE CONTROL

A definition of adaptive control that is widely accepted is (Åstrom and Wittenmark, 1990) a controller with adjustable parameters and a mechanism for adjusting the parameters. An adaptive controller has a distinct architecture, consisting of two loops: a control loop and a parameter adjustment loop.

Dell'Acqua P., Lombardi A. and Moniz Pereira L. (2006).
MODELLING ADAPTIVE CONTROLLERS WITH EVOLVING LOGIC PROGRAMS.
In *Proceedings of the Third International Conference on Informatics in Control, Automation and Robotics*, pages 107-112
DOI: 10.5220/0001220501070112

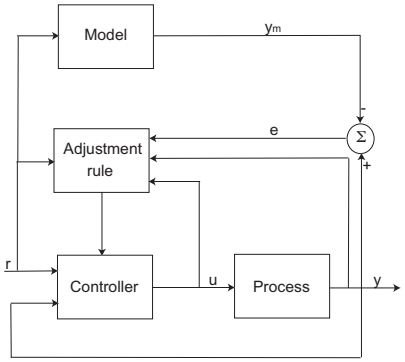

Figure 1: Scheme of a model reference adaptive system.

2.1 Model Reference Adaptive Control

In the model reference adaptive system (MRAS) (Åstrom and Wittenmark, 1990) the control is specified in terms of a reference model which tells how the process output ideally should respond to the command signal. A block diagram of the system is shown in Fig. 1.

The regulator system consists of two loops. An inner loop which is an ordinary feedback loop composed of the process and a controller. The parameters of the regulator are adjusted by the outer loop in such a way that the error e between the model output y_m and the process output y becomes small. This scheme is so-called direct approach because the adjustment rules tell directly how the regulator parameter should be updated.

The key problem is to determine the adjustment mechanism so that the process output becomes close to the model output making the error going to zero.

2.2 Fuzzy Control Systems

Fuzzy logic control (Feng et al., 1997) has proved to be a successful approach for complex nonlinear systems. In many cases it has been suggested as an alternative approach to conventional control techniques. Fuzzy logic control techniques represent a means of both collecting human knowledge and expertise and dealing with uncertainties in the process of control.

Fuzzy control usually decomposes the complex system into several subsystem according to the human expert's understanding of the system and uses a simple control law to emulate the human control strategy in each local operating region. The global control law is then constructed by combining all the local control actions through fuzzy membership functions.

Many physical systems are very complex in practice so that rigorous mathematical models can be very difficult if not impossible to obtain. However many physical systems can be expressed in some form of mathematical model locally, or as an aggregation of a set of mathematical models. The fuzzy dynamic model, proposed by Takagi and Sugeno (Takagi and Sugeno, 1985) is described by fuzzy IF-THEN rules which locally represent nonlinear systems. The following fuzzy model represents a complex single-input-single-output system and it includes rules and local analytic linear models:

$$\begin{aligned} R^i : \text{IF} \quad & z_1 \text{ is } F_1^i \text{ AND } \ldots \text{ AND } z_s \text{ is } F_s^i \\ \text{THEN} \quad & \dot{x}(t) = A_i x(t) + B_i u(t) \\ & y_i(t) = C_i x(t) \end{aligned} \tag{1}$$

where $i = 1, 2, \ldots, m$, R^i denotes the $i-th$ fuzzy inference rule, m the number of inference rules, $F_j^i (j = 1, 2, \ldots, s)$ are fuzzy sets, $x(t) \in \Re^n$ the system state variables, $u(t) \in \Re^p$ the system input variables, $y_i(t)$ and (A_i, B_i, C_i) the output and the matrix triple of the $i-th$ subsystem, and $z(t) = [z_1, z_2, \ldots z_s]$ some measurable system variables.

Let $\mu_i(x(t))$ be the normalized membership function of the inferred fuzzy set F^i where

$$F^i = \prod_{j=1}^{s} F_j^i, \qquad \sum_{i=1}^{m} \mu_i = 1 \tag{2}$$

then the final output $y(t)$ of the system is inferred by taking the weighted average of the outputs $y_i(t)$ of each subsystem, that is

$$y(t) = \sum_{i=1}^{m} \mu_i y_i(t) \tag{3}$$

It should be noted that the global fuzzy model is nonlinear time-varying since the membership functions are nonlinear and time-varying in general. The developed fuzzy model includes two kinds of knowledge: one is the qualitative knowledge represented by the fuzzy IF-THEN rules, and the other is the quantitative knowledge represented by the local dynamic models. The model has a structure of a two level control system with the lower level providing basic feedback control and the higher level providing supervisory control or scheduling. A basic idea of control is to design local feedback controllers based on local models and then construct the global controller from the local controllers.

In (Feng, 2002) an adaptive control design method for a class of fuzzy dynamic models has been proposed. The basic idea is to design an adaptive controller in each local region and then construct the global adaptive controller by suitably integrating the local adaptive controllers together in such a way that the global closed-loop adaptive control system is stable. Adaptive fuzzy control, often called self-organizing fuzzy control (SOC) (Mamdani and Baaklini, 1975), can be classified as a MRAS. It has a hierarchical structure in which the inner loop is a table

based controller and the outer loop is the adjustment mechanism. The idea behind self-organization is to let the adjustment mechanism update the values in the control table on the basis of the current performance of the controller.

3 EVOLVING LOGIC PROGRAMMING

In this section we recap the paradigm of Evolving Logic Programming (EVOLP), a simple though quite powerful extension of logic programming (Alferes et al., 2002)[1]. EVOLP allows to model the dynamics of knowledge bases expressed by programs, as well as specifications that dynamically change.

3.1 Language

To make a logic program evolve one needs some mechanism for letting older rules be supervened by more recent ones. That is, one must include a mechanism for deletion of previous knowledge. This can be achieved by permitting negation not just in bodies of rules, but in their heads as well[2]. Moreover, one needs a means to state that, under some conditions, some new rule is to be added to the program. In EVOLP this is achieved by augmenting the language with a reserved predicate $assert/1$, whose argument is itself a rule, so that arbitrary nesting becomes possible. This predicate can appear both as rule head (to impose internal assertions of rules) as well as in rule bodies (to test for assertion of rules).

In the following we let $\mathcal{L}$ be any propositional language not containing the predicate $assert/1$. Given $\mathcal{L}$, the extended language $\mathcal{L}^+$ is defined inductively as follows:

- All propositional atoms in $\mathcal{L}$ are propositional atoms in $\mathcal{L}^+$.
- If every $L_0, \ldots, L_n$ ($n \geq 0$) is a literal in $\mathcal{L}^+$ (i.e. a propositional atom A or its default negation $not\ A$), then $L_0 \leftarrow L_1, \ldots, L_n$ is a rule[3] over $\mathcal{L}^+$.
- If R is a rule over $\mathcal{L}^+$, then $assert(R)$ is a propositional atom of $\mathcal{L}^+$.
- Nothing else is a propositional atom in $\mathcal{L}^+$.

Given a rule $L_0 \leftarrow L_1, \ldots, L_n$, then L_0 is the head of the rule, and $L_1, \ldots, L_n$ is the body. Rules with empty body (that is, $n = 0$) are written as $L_0 \leftarrow$ and are called facts.

An *evolving logic program* over $\mathcal{L}$ is a (possibly infinite) set of rules over $\mathcal{L}^+$. Consider the two rules:

$$assert(not\, a \leftarrow b) \leftarrow not\, c$$
$$a \leftarrow assert(b \leftarrow)$$

Intuitively, the first rule states that, if c is false, then the rule $not\, a \leftarrow b$ must be asserted. The 2nd rule states that, if the fact $b \leftarrow$ is going to be asserted, then a is true.

The language $\mathcal{L}^+$ allows one to model the knowledge base self-evolution. What is needed now is a way to make the system aware of events that happen outside it, e.g., the observation of facts (or rules) that are perceived at some state or assertion commands imparting the assertion of new rules on the evolving program. Both observations and assertion commands can be represented as EVOLP rules: the former by rules without the assert predicate in the head, and the latter by rules with it. Thus, in EVOLP outside influence can be represented as a sequence of sets of EVOLP rules. This leads us to the following notion. An *event sequence* $\mathcal{E}$ over an evolving logic program P is a sequence of evolving logic programs over the language $\mathcal{L}$ of P.

3.2 Semantics

The meaning of a sequence of EVOLP programs is given by a set of *evolution stable models*, each of which is a sequence of interpretations. The basic idea is that each evolution stable model describes some possible evolution of one initial program after a given number n of evolution steps with respect to an event sequence $\mathcal{E}$.

The construction of these program sequences is as follows: whenever the atom $assert(Rule)$ belongs to an interpretation in a sequence, i.e. belongs to a model according to the stable model semantics of the current program, then $Rule$ must belong to the program in the next state; asserts in bodies are treated as any other predicate literals.

Program sequences are treated as in the framework of *dynamic logic program* (DLP). A dynamic logic program $P_1 \circ \cdots \circ P_n$ is a sequence of generalized logic programs (P_n being the most recent one). The idea of DLP is that the most recent rules (i.e., the ones belonging to the most recent programs in the sequence) are set in force, and previous rules are valid (by inertia) insofar as possible, i.e. they are kept for as long as they do not conflict with more recent ones. For the formal definition of the declarative and procedural semantics of DLP see (Alferes et al., 2000).

[1] An implementation of EVOLP is available from http://centria.fct.unl.pt/~jja/updates.

[2] A well known extension to normal logic programs (Lifschitz and Woo, 1992).

[3] Rules of the form $L_0 \leftarrow L_1, \ldots, L_n$, where each L_i is a literal, are typically called generalized logic programming rules.

Consider the program P:

$$a \leftarrow$$
$$assert(b \leftarrow a) \leftarrow not\, c$$
$$c \leftarrow assert(not\, a \leftarrow)$$
$$assert(not\, a \leftarrow) \leftarrow b$$

For simplicity suppose that all events in $\mathcal{E}$ are empty. The (only) stable model of P is $I = \{a, assert(b \leftarrow a)\}$ and it conveys the information that program P is ready to evolve into a new program $P \circ P_2$ by adding rule $(b \leftarrow a)$ at the next step, i.e. in P_2. In the only stable model I_2 of the new program $P \circ P_2$, atom b is true as well as atom $assert(not\, a \leftarrow)$ and also c, meaning that $P \circ P_2$ is ready to evolve into a new program $P \circ P_2 \circ P_3$ by adding rule $(not\, a \leftarrow)$ at the next step, i.e. in P_3. Now, the (negative) fact $not\, a \leftarrow$ in P_3 conflicts with the fact $a \leftarrow$ in P, and so this older fact is rejected. The rule added in P_2 remains valid, but is no longer useful to conclude b, since a is no longer valid. So, $assert(not\, a \leftarrow)$ and c are also no longer true. In the only stable model of the last sequence both a, b, and c are false.

This example simplifies the problem of defining the semantics in that it does not consider the influence of events from the outside. In fact, as stated above, all those events are empty. To take into consideration outside events, the rules that came in the i-th event are added to the program of state i. Suppose that at state 2 there is an event from the outside $E_2 = \{assert(d \leftarrow b) \leftarrow a;\ e \leftarrow\}$. Since the only stable model of P is $I = \{a, assert(b \leftarrow a)\}$ and there is an outside event at state 2, the program should evolve into the new program obtained by updating P not only with the rule $b \leftarrow a$ but also with the rules in E_2, i.e. $P \circ \{b \leftarrow a;\ assert(d \leftarrow b) \leftarrow a;\ e \leftarrow\}$. The only stable model I_2 of this program is now $\{a, b, e, assert(not\, a \leftarrow), assert(d \leftarrow b), assert(b \leftarrow a)\}$.

In EVOLP the rules coming from the outside, be they observations or assertion commands, are understood as events given at a certain state, but which are not to persist by inertia. That is, if a rule R belongs to an event E_i of an event sequence $\mathcal{E}$, then R was perceived after $i-1$ evolution steps of the program, and this perception event is not to be assumed by inertia from then onward. Thus, in the previous example, when constructing subsequent states, the rules coming from events in state 2 should no longer be available and considered. This understanding is formalized as follows.

An *evolution interpretation* of length n of an evolving logic program P over $\mathcal{L}$ is a finite sequence $\mathcal{I} = \langle I_1, I_2, \ldots, I_n \rangle$ of sets of propositional atoms of $\mathcal{L}^+$. The *evolution trace* associated with an evolution interpretation $\mathcal{I}$ is the sequence of programs $\langle P_1, P_2, \ldots, P_n \rangle$ where:

$$P_1 = P \; and \; P_i = \{R \mid assert(R) \in I_{i-1}\}$$

for each $2 \leq i \leq n$. An evolution interpretation of length n, $\langle I_1, I_2, \ldots, I_n \rangle$, with evolution trace $\langle P_1, P_2, \ldots, P_n \rangle$, is an *evolution stable model* of P given $\mathcal{E} = \langle E_1, E_2, \ldots, E_n \rangle$ iff for every i ($1 \leq i \leq n$), I_i is a stable model at state i of $P_1 \circ P_2 \cdots \circ (P_i \cup E_i)$.

Notice that the rules coming from the outside indeed do not persist by inertia. At any given state i, the rules from E_i are added to P_i and the (possibly various) stable models I_i are calculated. This determines the programs P_{i+1} of the trace, which are then added to E_{i+1} to determine the stable models I_{i+1}.

Being based on stable models, evolving logic programs may have various evolution stable models, as well as no evolution stable models at all. Consider the following program:

$$assert(a \leftarrow) \leftarrow not\, assert(b \leftarrow), not\, b$$
$$assert(b \leftarrow) \leftarrow not\, assert(a \leftarrow), not\, a$$

it has two evolution stable models of length 3 wrt. the event sequence $\mathcal{E} = \langle \emptyset, \emptyset, \emptyset \rangle$ of empty events. Each model represents one possible evolution of the program:

$$\langle \{assert(a \leftarrow)\}, \{a, assert(a \leftarrow)\}, \{a, assert(a \leftarrow)\} \rangle$$
$$\langle \{assert(b \leftarrow)\}, \{b, assert(b \leftarrow)\}, \{b, assert(b \leftarrow)\} \rangle$$

Given an evolution trace $\langle P_1, \ldots, P_n \rangle$ and an event sequence $\mathcal{E} = \langle E_1, \ldots, E_n \rangle$, a state i ($1 \leq i \leq n$) is *deterministic* iff $P_1 \circ P_2 \circ \cdots \circ (P_i \cup E_i)$ has a unique stable model. P is deterministic wrt. $\mathcal{E}$ iff every state i is deterministic.

Since various evolutions may exist for a given length, evolution stable models alone do not determine a truth relation. But one such truth relation can be defined, as usual, based on the intersection of models. One important case is when the program is stratified, for then there is a single stable model, and a deterministic evolution.

Given a program P and an event sequence $\mathcal{E}$ of length n, a propositional atom A is: *true* iff $A \in I_n$ for every evolution stable model $\langle I_1, I_2, \ldots, I_n \rangle$ of P and $\mathcal{E}$; *false* iff $A \notin I_n$ for every evolution stable model $\langle I_1, I_2, \ldots, I_n \rangle$ of P and $\mathcal{E}$; *unknown* otherwise.

4 MODELLING ADAPTIVE CONTROLLERS WITH EVOLP

The simplest way to model an adaptive logic-based controller is to employ EVOLP as the language of the controller as illustrated in Fig. 2. In this case, we have a discrete, adaptive controller governed by logic-based rules. This scheme can be seen as a simplified version of MRAS (see Fig. 1), where the adjustment rule block is composed of the logic rules

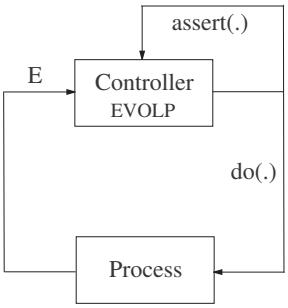

Figure 2: Adaptive logic-based controller.

defining the predicate *assert*. This makes it possible that old rules of the controller are replaced by new rules. The theory of the controller is formalized by an (initial) evolving logic program that can evolve via its (internal) assertion commands, and/or via the input to the controller received as events. The language of the controller contains distinguished atoms expressing the actions the controller wants to execute on the process/environment through its actuators. We assume that such atoms take the form *do(action)*. The intuition is that whenever an atom *do(action)* is true at the current state, the corresponding action is performed by the actuator on the process. Then, the output of the process is received by the controller in the form of facts (or more generally rules) represented as events. At each state n, the controller considers all the stable models of $P_1 \circ P_2 \circ \cdots \circ (P_n \cup E_n)$, and (i) sends to the actuators the actions corresponding to all the atoms *do(action)* that are true at n, and (ii) self-evolves by considering all the atoms $assert(R)$ that are true.

Example: a controller for a lift
Consider the controller of a lift that receives from outside signals of the form $push(N)$, when somebody pushes the button for going to floor N, or *floor*, when the lift reaches a new floor. Upon receipt of a *push(N)* signal, the lift records that a request for going to floor N is pending:

$$assert(\textit{request}(F)) \leftarrow \textit{push}(F)^4$$

Mark the difference between this rule and the rule $request(F) \leftarrow push(F)$. When the button F is pushed, with the latter rule $request(F)$ is true only at that moment, while with the former $request(F)$ is asserted to the evolving program so that it remains inertially true (until its truth is possibly deleted afterwards).

Based on the pending requests at each moment, the controller must prefer where to go:

$$\begin{aligned}
&\textit{going}(F) \leftarrow \textit{request}(F), not\ \textit{unpref}(F),\\
&\qquad\qquad not\ \textit{fire_alarm}\\
&\textit{unpref}(F) \leftarrow \textit{request}(F2), \textit{better}(F2, F)\\
&\textit{better}(F1, F2) \leftarrow \textit{at}(F),\ |F1 - F| < |F2 - F|\\
&\textit{better}(F1, F2) \leftarrow \textit{at}(F),\ |F1 - F| = |F2 - F|,\\
&\qquad\qquad F2 < F
\end{aligned}$$

Notice that the body of the first rule above contains the non-monotonic condition, not *fire_alarm*. This is needed to stop the lift (i.e., it makes *going*(F) false) in situations where the event *fire_alarm* is received from the outside. Predicate $at/1$ stores, at each moment, the number of the floor where the lift is located. Thus, if a *floor* signal is received, depending on where the lift is going, *at*(F) must be incremented (resp. decremented):

$$\begin{aligned}
&assert(\textit{at}(F+1)) \leftarrow \textit{floor}, \textit{at}(F), \textit{going}(G),\ G > F\\
&assert(not\ \textit{at}(F)) \leftarrow \textit{floor}, \textit{at}(F), \textit{going}(G),\ G > F
\end{aligned}$$

When the lift reaches the floor to which it was going, it must open the door. After that, it must remove the pending request for going to that floor:

$$\begin{aligned}
&\textit{do}(\textit{open}(F)) \leftarrow \textit{going}(F), at(F)\\
&assert(not\ \textit{request}(F)) \leftarrow \textit{going}(F), at(F)
\end{aligned}$$

Modelling uncertainty
Consider a scenario where one wants to formalize a controller that monitors the room temperature and consequently activates a heating device to maintain the temperature within some specified range. Assume that the controller receives contradictory/uncertain data from its sensors, e.g., it contemporaneously receives two distinct temperatures x and y whose difference is greater than a specified value. The behaviour of the controller can be defined in such a situation by a simple rule of the form:

$$\textit{do(action)} \leftarrow \textit{tempA}(x), \textit{tempB}(y), x > y + 5$$

In other situations, the controller may have incomplete knowledge of the outside environment. Suppose that in the lift example above, the controller receives a signal that may (or may not) be a *floor* signal. This can be coded as an event consisting of two rules:

$$\begin{aligned}
&\textit{floor} \leftarrow not\ \textit{no_signal}\\
&\textit{no_signal} \leftarrow not\ \textit{floor}
\end{aligned}$$

At this state there are 2 stable models: one corresponding to the evolution in case the floor signal is considered; the other, in case it isn't. The truth relation can here be used to determine what is certain despite the undefinedness of the events received, i.e., true in every stable model, e.g., what may be triggered if *fire_alarm* is also received: *do*$(stop) \leftarrow$ *fire_alarm*.

Properties of the system
Since the controller is axiomatized by logic rules, it

[4]Rules with variables stand for all their ground instances.

is possible to formally prove a number of properties. Typically, such properties take one of the following two forms. Let P be the (initial) evolving logic program that axiomatizes the theory of the controller, and $\mathcal{E}$ a (finite) event sequence representing the input to the controller.

$$\forall \mathcal{E}\ \exists n.\ \textit{Property} \qquad \textit{(weak)}$$
$$\forall \mathcal{E}\ \forall n.\ \textit{Property} \qquad \textit{(strong)}$$

Reconsider the example of the lift controller. Then, one can guarantee: (i) the safety condition that the lift will never open its door if it is not at some floor by proving that:

$$\forall \mathcal{E}\ \forall n\ \forall x.\ not\ (\textit{open}(x)\ \wedge\ not\ \textit{at}(x))$$

or (ii) the fairness condition that if the button of a certain floor has been pushed, then the lift will eventually go to that floor by proving that:

$$\forall \mathcal{E}\ \exists n\ \forall x.\ \textit{push}(x) \supset \textit{at}(x)$$

It is easy to prove that the above property does not hold if the policy to handle the pending requests is the one axiomatized by the rules for *going*/1.

5 CONCLUSION

In this paper we have addressed the problem of modelling adaptive logic-based controllers by means of Evolving Logic Programs. One advantage of using a well-defined, logic-based approach is that it is possible to formally prove properties of the controller. Moreover, various forms of logic reasoning (e.g., abduction, hypothetical reasoning, rule mining) can be integrated into the logic framework and employed to enhance the controller's performance in cases where there is uncertainty due to the complexity of the environment.

The use of abduction, a well-developed technique in the Logic Programming paradigm, will enable us to diagnose erroneous controller behaviour, by automatically hypothesizing possible faults. Furthermore, abduction can be employed to prove correctness of a controller specification, by showing that no physically meaningful hypothesized sequence of events can result in some integrity violation by the controller, as in the approach in (de Castro and Pereira, 2004).

Since the semantics of EVOLP is stable model based, it is possible to characterize uncertainty by having at a certain state several stable models. This corresponds to the case where there exist branches in the evolution stable model of the program. Clearly, it is possible to guarantee that the program will evolve into a unique branch by enforcing syntactic restrictions on programs. In fact, if the program is stratified then there will be only one stable model, and therefore no branching can occur. This hypothesis is however unrealistic in most of the cases. A better solution to the problem would be to exploit preference reasoning in order to prefer among alternatives when a branching situation occur. This is possible since preference reasoning can be employed to prefer among alternative stable models (Alferes and Pereira, 2000). Moreover, preferences themselves are updatable, and this empowers a form of meta-control.

Additionally, EVOLP can be used to simulate possible futures, and then preferences may be used to chose desired futures or to avoid undesirable ones. This means one can have lookahead proactive control.

REFERENCES

Alferes, J. J., Brogi, A., Leite, J. A., and Pereira, L. M. (2002). Evolving logic programs. In *Procs. 8th European Conf. on Logics in Artificial Intelligence (JELIA'02)*, vol. 242 of *LNAI*, pp. 50–61.

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

Alferes, J. J. and Pereira, L. M. (2000). Updates plus preferences. In Aciego, M. O., de Guzmn, I. P., Brewka, G., and Pereira, L. M., editors, *Logics in AI, Procs. JELIA'00*, LNAI 1919, pp. 345–360.

Åstrom, K. J. and Wittenmark, B. (1990). *Computer-Controlled Systems.Theory and Design*. Prentice Hall Internal Inc.

Castro, J. F. and Pereira, L. M. (2004). Abductive validation of a power-grid expert system diagnoser. In *Procs. 17th Int. Conf. on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA-AIE'04)*, vol. 3029 of *LNAI*, pp. 838–847.

Feng, G. (2002). An approach to adaptive control of fuzzy dynamic systems. *IEEE Transactions on Fuzzy Systems*, 10(2):268–275.

Feng, G., Cao, S., Rees, N., and Cheng, C. (1997). Analysis and design of model based fuzzy control systems. In *Proc. Sixth IEEE Int. Conf. on Fuzzy Systems*, vol. 2, pp. 901 – 906.

Hespanha, J. P., Liberzon, D., and Morse, A. S. Overcoming the limitations of adaptive control by means of logic-based switching. Systems and Control Letters, 49(1):49-65.

Lifschitz, V. and Woo, T. (1992). Answer sets in general non-monotonic reasoning (preliminary report). In Nebel, B., Rich, C., and Swartout, W., editors, *KR'92*. Morgan-Kaufmann.

Mamdani, E. and Baaklini, N. (1975). Prescriptive method for deriving control policy in a fuzzy-logic controller. *Electronic Letters*, 11(25/26):625–626.

Takagi, T. and Sugeno, M. (1985). Fuzzy identification of systems and its applications to modeling and control, *IEEE Trans. Syst. Man. & Cybern.*, 15(1):116–132.