An Evolvable Rule-Based E-mail Agent

J. J. Alferes[1], A. Brogi[2], J. A. Leite[1], and L. M. Pereira[1]

[1] CENTRIA, Universidade Nova de Lisboa, Portugal
[2] Dipartimento di Informatica, Università di Pisa, Italy

Abstract. The Semantic Web is a "living organism", which combines autonomously evolving data sources/knowledge repositories. This dynamic character of the Semantic Web requires (declarative) languages and mechanisms for specifying its maintenance and evolution. For example, for changing the behaviour of a data source, so that a new rule becomes into effect, one should not be concerned with the complex, interrelated, and dynamically obtained knowledge, and should have a way to simply specify what knowledge is to be changed. This requires the existence of a language for exacting such changes (or updates), which takes in consideration the addition/deletion and changes of rules, thereby automating the task of dealing with inconsistencies arising from those updates. To address this issue, we resort to recent developments in the field of Logic Programming, and show how the framework of EVOLP (EVOlving Logic Programs) can be put to work to model such reactive and updateable rule bases, bringing an important added value to RuleML. We make our case by exhibiting a detailed application example of how EVOLP can be used to express updateable RuleML rule bases, employing it to define an evolving e-mail Personal Assistant Agent.

1 Introduction

The World Wide Web (WWW) is, without a doubt, a great success story, bringing new challenges and opportunities at almost every conceivable level of our existence. While its exponential growth was initially seen as one of its bigger attractive features, it now contributes to one of its major challenges: provide structure and organization to its contents, thus shifting its data and documents from the current human oriented format to a more machine understandable one where a greater level of automation can be achieved. The World Wide Web Consortium's (W3C) [7] Semantic Web Activity [1] was launched to address these concerns. While the eXtensible Markup Language (XML) [20] provides a method to structure documents, and the Resource Description Framework (RDF) [17] one for exposing their meaning, it is widely accepted now the need to reason with, and about, WWW documents. Rules provide the natural and widely accepted mechanism to perform automated reasoning, with available mature theory and technology. The Rule Markup Initiative (RuleML) [13] aims at defining a core format for rule interchange, resorting to XML.

The Logic Programming paradigm, with its well-defined, general, integrative, encompassing, and rigorous framework, together with its recent technolog-

ical and theoretical developments, brings new insights to the above issues, and provides inspiration to this general promising area of investigation [10, 11, 18].

The Semantic Web is a "living organism", which combines autonomously evolving data sources/knowledge repositories. This dynamic character of the Semantic Web requires (declarative) languages and mechanisms for specifying its maintenance and evolution. For example, for changing the behaviour of a data source, so that a new rule becomes into effect, one should not per force have to be concerned with the complex, interrelated, and dynamically obtained knowledge, and should have a way to simply specify what knowledge is to be changed. This requires the existence of a language for exacting such changes (or updates), which takes in consideration the addition/deletion and changes of rules, thereby automating the task of dealing with inconsistencies arising from those updates. To address this issue, we resort to recent developments in the field of Logic Programming, and show how the framework of EVOLP [2] can be put to work to model such reactive and updateable rule bases, in effect bringing an important added value to RuleML.

The EVOLP framework appeared in the line of development of other existing languages (such as LUPS [4], EPI [8] and KABUL [14]) for specifying and programming the evolution of knowledge bases represented as logic programs. Distinctly from these extant languages, which introduce a lot of new programming constructs, each encoding a high level behaviour of addition and deletion of rules, EVOLP's up front goal was to enable with the evolution of logic programs by adding to traditional logic programming as few constructs as possible, i.e. by staying as close as possible to the usual language of logic programs.

EVOLP can adequately express the semantics resulting from successive updates to logic programs, considered as incremental specifications of knowledge bases, and whose effect may be contextual. It automatically and appropriately deals, via its update semantics (based on Dynamic Logic Programming [3]), with possible potential contradictions arising from successive specification changes and refinements. Furthermore, the EVOLP language can express self-updates triggered by the evolution context itself, present or future. Additionally, foreseen updates not yet executed can automatically trigger other updates, and moreover updates can be nested, so as to (contextually) determine change, both in the next state and in other states further down an evolution strand.

It is the goal of this paper to show that the attending formulation of EVOLP provides a good firm formal basis in which to express, implement, and reason about dynamic RuleML rule bases. To do this, in the ensuing section we recap the formal syntax and semantics of EVOLP. Immediately afterwards we make our case by exhibiting a detailed and protracted application example of how EVOLP can be mustered to express updateable RuleML rule bases, employing it to define an evolving e-mail Personal Assistant Agent, whose executable specification evolves by means of external and of internal dynamic updates, both of which can be made contingent on the evolution context in which they occur. We terminate with a section comprising discussion, comparisons with related application work, open issues, and themes for future development.

2 Evolving logic programs

In this section we briefly recall the language and semantics of EVOLP [2]. EVOLP is a logic programming language that caters for the evolution of an agent's knowledge, be it caused by external events, or by internal requirements for change. Moreover, it does so by adding as few new constructs to traditional logic programming as possible.

2.1 Language

What is required to let logic programs evolve? For a start, 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 to accompany the agent's knowledge evolution. This can be achieved by permitting default negation not just in rule bodies, as in normal logic programming, but in rule heads as well[1].

Moreover, one needs a means to state that, under some conditions, some new rule or other is to be added to the program. This is achieved in EVOLP simply by augmenting the language with a reserved predicate $assert/1$, whose sole argument is itself a full-blown rule, and so 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). Formally:

Definition 1. *Let $\mathcal{L}$ be any propositional language (not containing the predicate $assert/1$). The* extended language *$\mathcal{L}_{assert}$ is defined inductively as follows:*

- *All propositional atoms in $\mathcal{L}$ are propositional atoms in $\mathcal{L}_{assert}$;*
- *If each of $L_0, \ldots, L_n$ is a literal in $\mathcal{L}_{assert}$ (i.e. a propositional atom A or its default negation not A), then*

$$L_0 \leftarrow L_1, \ldots, L_n$$

  is a generalized logic program rule over $\mathcal{L}_{assert}$ (and dubbed EVOLP rule);
- *If R is a rule over $\mathcal{L}_{assert}$ then $assert(R)$ is a propositional atom of $\mathcal{L}_{assert}$;*
- *Nothing else is a propositional atom in $\mathcal{L}_{assert}$.*

An evolving logic program *over a language $\mathcal{L}$ (or EVOLP program, for short) is a (possibly infinite) set of generalized logic program rules over $\mathcal{L}_{assert}$.*

Besides internal updates, which may be modelled with the above, EVOLP allows too for influence from the outside, where this influence may be: observation of facts (or rules) that are perceived at some state; assertion commands directly imparting the assertion of new rules on the evolving program. Both can be represented as EVOLP rules: the former by rules without the assertion predicate in the head, and the latter by rules having it. Accordingly, outside influence is represented as a sequence of EVOLP programs:

Definition 2. *Let P be an evolving program over the language $\mathcal{L}$. An* event sequence *over P is a sequence of evolving programs over $\mathcal{L}$.*

[1] A known extension to normal logic programs [15], named generalized logic programs.

2.2 Semantics

In general, an EVOLP program describes an agent's initial knowledge base. This knowledge base may already include rules (sporting asserts in heads) that describe the forms of its own evolution. Besides, EVOLP considers sequences of events representing observations and commands arising from the outside. Each event in the sequence is itself a set of EVOLP rules, i.e. an EVOLP program. Thus the semantics issue is that of, when given an initial EVOLP program and a sequence of EVOLP programs as events, to determine what is true and what is false after each event. More precisely, the meaning of a sequence of EVOLP programs is afforded by a set of *evolution stable models*, each of which being a sequence of interpretations or states. The basic compelling idea is that each evolution stable model describes a possible evolution of the initial program along a given number n of evolution steps, taking into account the events in the sequence. Each evolution is represented by a sequence of programs, each corresponding to a knowledge state.

The primordial intuitions for the construction of these program sequences are as follows: Regarding head asserts, whenever the atom $assert(Rule)$ belongs to an interpretation in a given sequence, i.e. belongs to that model according to the stable model semantics of the current program, then $Rule$ must belong to the program in the next state; Moreover, assertion literals in bodies are treated like any other predicate literals. Furthermore, to deal with outside events, i.e. with sequences of EVOLP programs, rules in the i-th event are added on one occasion only to the program of state i.

Sequences of programs are then treated as in DLP (Dynamic Logic programming) [3], where the most recent rules 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. Default negation is treated as in stable models of normal [9] and generalized programs [15]. Formally, a *dynamic logic program* is a sequence $P_1 \oplus \ldots \oplus P_n$ (also denoted $\bigoplus \mathcal{P}$, where $\mathcal{P}$ is a set of generalized logic programs indexed by $1, \ldots, n$), and its semantics is determined by[2]:

Definition 3. *Let $\bigoplus\{P_i : i \in S\}$ be a dynamic logic program over language $\mathcal{L}$, let $s \in S$, and let M be a set of propositional atoms of $\mathcal{L}$. Then:*

$$Default_s(M) = \{not\, A \leftarrow . \mid \nexists A \leftarrow Bdy \in P_i (1 \leq i \leq s) : M \models Bdy\}$$
$$Reject_s(M) = \{L \leftarrow Bdy \in P_i \mid \exists\, not\, L \leftarrow Bdy' \in P_j, i < j \leq s,\ M \models Bdy'\}$$

where A is an atom, $not\, L$ denotes the complement w.r.t. default negation of the head literal L, and both Bdy and Bdy' are conjunctions of literals. $M \models Bdy$ iff for every atom A (resp. literal $not\, A$) in Bdy, $A \in M$ (resp. $A \notin M$).

Definition 4. *Let $\mathcal{P} = \bigoplus\{P_i : i \in S\}$ be a dynamic logic program over language $\mathcal{L}$. A set M of propositional atoms of $\mathcal{L}$ is a stable model of $\mathcal{P}$ at state $s \in S$ iff:*

$$M' = least\left(\left[\bigcup\nolimits_{i \leq s} P_i - Reject_s(M)\right] \cup Default_s(M)\right)$$

[2] For more details on dynamic logic programming the reader is referred to [3].

where $M' = M \cup \{not\_A \mid A \notin M\}$*, and* $least(.)$ *denotes the least (wrt. set inclusion) model of the definite program obtained from the argument program by replacing every default negated literal* $not\, A$ *by a new atom* not_A*.*

Definition 5. *An* evolution interpretation *of length* n *of an evolving 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* $\mathcal{L}_{assert}$*. 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* $2 \leq i \leq n$*.*

Definition 6. *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 a sequence of events* $\langle E_1, E_2, \ldots, E_k \rangle$*, with* $n \leq k$*, iff for every* i $(1 \leq i \leq n)$*,* I_i *is a stable model at state* i *of* $P_1 \oplus P_2 \ldots \oplus (P_i \cup E_i)$*. We say that an atom* A *is true (resp. false) after* n *steps if* A *belongs (resp. does not belong) to the last interpretation of all the (resp. any of the) evolution stable models of length* n*.*

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

To better understand the definitions, we put them to work on an example.

Example 1. Consider the initial program P:

$$a. \quad assert(b \leftarrow a) \leftarrow not\, c. \qquad c \leftarrow assert(not\, a \leftarrow). \quad assert(not\, a \leftarrow) \leftarrow b, d.$$

stating that: a is true; whenever c is false, rule $b \leftarrow a$ is to be asserted; whenever fact $not\, a$ is to be asserted, c is true; and if b and d are true then fact $not\, a$ is to be asserted. Moreover, suppose the following sequence of events was perceived:

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

The (only) stable model of $P \cup E_1$ is $I_1 = \{a, assert(b \leftarrow a), assert(d \leftarrow a)\}$ and it conveys the information that program P is ready to evolve into a new program $P \oplus P_2$ by adding rules $(b \leftarrow a)$ and $(d \leftarrow a)$ at the next step, i.e. in P_2. In the only stable model I_2 of the new program $P \oplus (P_2 \cup E_2)$, atoms e, b and d are true, as well as atoms $assert(not\, a \leftarrow)$ and c. Thus, $P \oplus P_2$ is ready to evolve into a new program $P \oplus P_2 \oplus P_3$ by adding rule $(not\, a \leftarrow)$ at the next step, i.e. in P_3. Now the negative fact in P_3 conflicts with the positive fact in P, and this older one is rejected. The rules added in P_2 remain valid, but are no longer useful to conclude b and d since a is no longer valid. So $assert(not\, a \leftarrow)$ and c are also no longer true. In the only stable model last in the sequence, I_3, all of a, b, and c are false. Moreover, there being no information on e on any of P_1, P_2 or P_3, e becomes false too. Accordingly, the only evolution stable model is $\langle I_1, I_2, I_3 \rangle$ with the I_is as described above.

Notice that the truth of the event e in the 2nd state does not persist by inertia to later states. Intuitively, in EVOLP, rules coming from the outside, be

they observations or assertion commands, are to be understood as events at a state, and so do not persist by inertia. The same holds for the truth of event $assert(d \leftarrow a)$ which also does not persist. However, its truth in the 1st state caused the addition of rule $d \leftarrow a$ to P_2, which itself persists therefrom as per the semantics of DLP.

3 An evolving E-mail Agent on the Web

Forthwith, EVOLP is employed to specify several features of a Personal Assistant agent for e-mail management on the Web, able to perform a few basic actions such as sending, receiving, and deleting messages, as well as moving them between folders, and to perform tasks such as filtering spam, storing messages in appropriate folders, sending automatic replies, notifying the user, and/or automatically forwarding specific messages, all of which dependant on user specified updatable criteria.

The specification of the policies that trigger agent actions can in general be modelled by a set of logic programming rules, and represented at a Web page by resorting to a Rule Markup Language (RuleML) with reaction rules triggering the actions. This way, all of the interface between the user, the knowledge base with the email messages, and the incoming messages could be achieved via this RuleML rule base: incoming messages would be treated as events given to the RuleML rule base via some Web service, and the user could access the messages via Web services querying the RuleML rule base. All the user would have to do would be to specify his policies in or translate them into RuleML.

If we expect the user to specify once and for all a consistent set of policies, then such a RuleML with reaction rules setting would be all that is required. But reality tells us otherwise: one observes that the user, every now and then, will discover new conditions under which incoming messages should be deleted, and under which messages now being deleted should be kept. If one allows dynamic specification of both the positive rule instances (e.g. *should be deleted*) and negative ones (e.g. *should not be deleted*) of such policies, soon the union of all rules becomes inconsistent. And one cannot expect the user to debug the RuleML rule base so as to invalidate all the old rules that no longer should be available due to more recent countervening ones. We would rather allow the user to simply state whatever new is to be enforced, and let the inference engine associated with the rule base automatically determine the old rules which may persist and those that do not. We are not even presupposing the user is contradictory, but just that he has updated his profile, something quite reasonable. For example, suppose he is tired of receiving spam messages advertising credit and states that all incoming messages whose subject contains the word *credit* are to be deleted. Later he finds out that important messages from his accountant are being deleted because the subject mentions *credit*. He should simply state that such incoming messages from his accountant are not to be deleted, and the inference mechanisms themselves should automatically determine that

such messages are not to be deleted, in spite of the previous rule. But if we just evaluate over the union of all specified policies, a contradiction is obtained.

It would also be important for the personal e-mail assistant agent to allow the specification of tasks not so simple as just performing actions whenever their conditions are met. Suppose one is organizing a conference and wants to automate part of the communication with referees and with authors. Basic tasks include automatic replies to authors whenever abstracts are submitted, etc. But more complex tasks can be conceived that we wish the agent to handle, such as: waiting for messages from referees about accepting to review a paper and, once they arrive, forwarding to him a message with the paper if it has already arrived, otherwise waiting till it arrives and then forwarding it; having different policies to deal with papers before and after the deadline; permitting the specification of extensions to the deadline on a case by case manner, and dealing differently with each paper; updating the initial specification for such policies; etc.

Next we show how an EVOLP based markup language, and corresponding inference mechanisms, would deal with the above mentioned contradictions, and automatically solves them with clear and precise semantics, as well as would account for the aforesaid more complex tasks. In the exposition, we concentrate on those features directly concerned with the evolving specification, namely the representation of the dynamic user profile, and of the dynamic specification of its action preconditions and their effects. Methods to specify other common simple tasks can easily be guaged from the exposition.

3.1 The general EVOLP/RuleML setting

To be able to specify RuleML rule bases using EVOLP as semantics, one first needs some XML schema to write EVOLP rules. This can easily be accomplished and, for the sake of readability, instead of defining that, and using XML notation for writing predicates and rules, here we simply write them as usual in logic programming. The EVOLP framework assumes that sequences of events are given to the evolving knowledge base, each of which is an EVOLP program. Each event causes the addition of an extra program in an evolving sequence $P_1 \oplus \ldots \oplus P_i$ of programs. This means our EVOLP rule bases must be able to cater for sequences of sets of rules, rather than simply for sets of rules. Thus, an appropriate tag (say, `<newProgram/>`) must exist in our schema in order to specify where new programs in the sequence are inserted [6]. We also need some mechanism for sending events, and for adding the new program to the rule base. This is performed via an appropriate Web service (whose description is outside the scope of this paper) that: receives the events; calls the EVOLP inference engine over the existing RuleML rule base in order to compute the new resulting program; and edits the rule base by adding to it the new program in the evolving sequence. Queries to the rule base are also posted via Web services that call the underlying EVOLP inference mechanism.

Our email agent is to be implemented via one such RuleML rule base with the specification of the user policies for his email. New incoming messages are simply viewed as events to the rule base, containing all the contents of the

message. More precisely, $newmsg\,(MsgId, From, Subject, Body)$ event are used for incoming message. As for outgoing messages, we presume that a Web service periodically queries the RuleML specification for true predicates of the form $send\,(To, Subject, Body)$, and dispatches corresponding messages.

To simplify the exposition, instead of having the policies stored in a different rule base from that of the (received) messages, we assume that both will be stored in the same rule base. Accordingly, messages will be stored as facts in the same rule base where the policy specifications sit. More precisely, for every message there will be a fact $msg\,(MsgId, From, Subject, Body, Time)$, along with a fact $in\,(MsgId, Folder)$ to represent that the message is in folder $Folder$. Viewing the contents of folders and messages then amounts to appropriately querying the rule base with the specification and messages for those predicates, and displaying the result via a transformed XML page.

3.2 One concrete example of policy specification

Now we show a concrete example of an email agent specification, and its evolution. We start with a very simple specification stating: all incoming messages are to be stored unless their deletion is explicitly specified; they are to be stored in the inbox folder unless otherwise stated to be automatically moved elsewhere; messages can be moved between folders, and deleted from folders. All this can be represented by the EVOLP program containing rules r_1 through r_{10} below, i.e. $P = \{\langle r_1\rangle, \langle r_2\rangle, \ldots, \langle r_{10}\rangle\}$.

$r_1 : time\,(1) \leftarrow$ $\qquad$ $r_2 : assert\,(time\,(T+1)) \leftarrow time\,(T)$
$r_3 : assert\,(not\, time\,(T)) \leftarrow time\,(T)$
$r_4 : assert\,(msg\,(M,F,S,B,T)) \leftarrow newmsg\,(M,F,S,B)\,, time\,(T)\,, not\, delete\,(M)$
$r_5 : assert\,(in\,(M, inbox)) \leftarrow newmsg\,(M,\_,\_,\_)\,, not\, move\,(M,F)\,, not\, delete\,(M)$
$r_6 : assert\,(in\,(M,F_{to})) \leftarrow newmsg\,(M,\_,\_,\_)\,, move\,(M,F_{to})$
$r_7 : assert\,(in\,(M,F_{to})) \leftarrow move\,(M,F_{from},F_{to})\,, in\,(M,F_{from})$
$r_8 : assert\,(not\, in\,(M,F_{from})) \leftarrow move\,(M,F_{from},F_{to})\,, not\, in\,(M,F_{to})$
$r_9 : assert\,(not\, in\,(M,F)) \leftarrow delete\,(M)\,, in\,(M,F)$
$r_{10} : assert\,(sent\,(To,S,B,T)) \leftarrow send\,(To,S,B)\,, time(T)$

The first three rules encode a clock, from now used to time-stamp all incoming messages. Such time-stamping is not really required, and an absolute (system) time could be used instead, but it is useful to show how a local clock can be encoded in EVOLP. Rule r_4 specifies that all incoming messages, when not specified to be deleted, indicated by literal $not\, delete\,(MsgId)$, are to be time-stamped and asserted as a fact along with the message. Rule r_5 specifies that all incoming messages, when not specified to be deleted nor specified to be moved to a specific folder, are to be stored in the folder $inbox$. Rule r_6 specifies the effect of moving an incoming message to a specific folder. Rules r_7 and r_8 encode the effect of moving a message between folders, represented by $move\,(MsgId, Folder_{from}, Folder_{to})$. Note no problem arises in specifying a message to be moved from and to the same folder. Rule r_9 specifies the effect of a delete action, represented by $delete\,(MsgId)$. This action causes a message to

be removed from its current folder. Finally, rule r_{10} encodes that sending a message causes the message to have been sent, hereby represented by the assertion of fact $sent\,(To, Subject, Body, Time)$.

At the initial state the stable model contains only $\{time\,(1)\,, assert(time(2)), assert(not\,time(1))\}$. With this initial specification, and since we do not yet have any rules to specify which incoming messages are to be deleted and which are to be moved, every received message is to be moved to folder *inbox*. Also, at every state transition, the clock increases its value. Suppose an event E_1 is received concerning three incoming messages:

$$newmsg\,(1, \text{“a@a”}, \text{“credit”}, \text{“some spam text”})$$
$$newmsg\,(2, \text{“accountant@c”}, \text{“hello”}, \text{“some text”})$$
$$newmsg\,(3, \text{“b@d”}, \text{“freecredit”}, \text{“more spam”})$$

After this event the stable model contains:

$$assert\,(msg\,(1, \text{“a@a”}, \text{“credit”}, \text{“some spam text”}, 1))\,, assert\,(in\,(1, inbox))$$
$$assert\,(msg\,(2, \text{“accountant@c”}, \text{“hello”}, \text{“some text”}, 1))\,, assert\,(in\,(2, inbox))$$
$$assert\,(msg\,(3, \text{“b@d”}, \text{“free credit”}, \text{“more spam”}, 1))\,, assert\,(in\,(3, inbox))$$
$$time\,(1)\,, assert\,(not\,time\,(1))\,, assert\,(time\,(2))$$

From this, program P_2 contains:

$$msg\,(1, \text{“a@a”}, \text{“credit”}, \text{“some spam text”}, 1)\,, in\,(1, inbox)\,, not\,time\,(1)$$
$$msg\,(2, \text{“accountant@c”}, \text{“hello”}, \text{“some text”}, 1)\,, in\,(2, inbox)$$
$$msg\,(3, \text{“b@d”}, \text{“free credit”}, \text{“more spam”}, 1)\,, in\,(3, inbox)\,, time\,(2)$$

indicating that the specification has been updated so as to store all messages, properly time-stamped, in folder *inbox*. Moreover the clock has been updated to its new value.

At this point, the user becomes upset with all the spam messages being received and decides to start deleting them on arrival. With this purpose, he updates the specification by asserting a general rule stating that spam messages should be deleted, encoded as the assertion of rule r_{11}, and he also updates the agent with a definition of what should be considered as spam, in this case simply those whose subject contains the word “credit”, encoded by the assertion of r_{12}.

$$r_{11} : delete\,(M) \leftarrow newmsg\,(M, F, S, B)\,, spam\,(F, S, B)$$
$$r_{12} : spam\,(F, S, B) \leftarrow contains\,(S, \text{“credit”})$$

Throughout, by definition consider literal $contains\,(S, T)$ true whenever T is contained in S, whose specification we omit for brevity. The assertion of these two rules, together with an update so as to delete messages 1 and 3, constitutes event $E_2 = \{assert\,(\langle r_{11}\rangle)\,, assert\,(\langle r_{12}\rangle)\,, delete\,(1)\,, delete\,(3)\}$. After this event, the stable model contains:

$$assert\,(\langle r_{11}\rangle)\,, assert\,(\langle r_{12}\rangle)\,, delete\,(1)\,, delete\,(3)\,, assert\,(not\,in\,(1, inbox))$$
$$assert\,(not\,in\,(3, inbox))\,, assert\,(time\,(3))\,, assert\,(not\,time\,(2))$$

together with those propositions of the form $msg/5$, $time/1$, $in/2$, representing the existing messages, their locations, and the current internal time[3].

From this model program P_3 is constructed, containing r_{11}, r_{12}, together with the facts $time\,(3)$, $not\,time\,(2)$, $not\,in\,(1, inbox)$ and $not\,in\,(3, inbox)$.

Suppose event E_3 is then received with:

$$newmsg\,(4, \text{“d@a”}, \text{“free credit card”}, \text{“spam spam spam”})$$
$$newmsg\,(5, \text{“accountant@c”}, \text{“credit”}, \text{“got your credit”})$$
$$newmsg\,(6, \text{“girlfriend@d”}, \text{“hi”}, \text{“theater tonight?”})$$

After this update, the stable model contains $spam(F, \text{“free credit card”}, \text{B})$, $assert(in(6, inbox))$, $assert(msg(6, \text{“girlfriend@d”}, \text{“hi”}, \text{“theater tonight?”}, 3))$, $spam(F, \text{“credit”}, \text{B})$, $delete(4)$, and $delete(5)$.

Since messages 4 and 5 are considered spam messages, they are both set for deletion and thus are not asserted. Only message 6 is asserted. From this model we construct the program P_4 which contains facts $not\,time\,(3)$, $in\,(6, inbox)$, $time\,(4)$, and $msg\,(6, \text{“girlfriend@d”}, \text{“hi”}, \text{“theater tonight?”}, 3)$.

Next we receive an event E_4 containing a single message[4]: $newmsg(7, \text{“accountant@c”}, \text{“are you there?”}, \text{“...”})$. This message makes the user aware that previous messages from his accountant had been deleted as spam. He then decides to update the definition of spam, to the effect that messages from his accountant are not spam. He achieves this by sending in an event asserting rule r_{13} (below). Note this rule is contradictory with rule r_{12}, for any message from the accountant with the subject containing the word “credit”. But EVOLP automatically detects such contradictions and resolves them by taking the newer rule to be an update of any previously existing ones, and we thus expect all such messages not to be deleted. Next the user is appointed conference chair and decides to update his email policy specification to perform some attending tasks. Henceforth, messages with the subject “abstract” are to be moved to folder *abstracts*, encoded by rule r_{14}, those containing the word “cfp” in their subjects should be moved to folder cfp (r_{15}). Furthermore, as the user is accustomed to only looking at his inbox folder, he wishes to be notified whenever an incoming message is immediately stored at a folder other than *inbox*. This is accomplished with rule r_{16}, which renders $notify\,(M)$ true in every such. Mark that $notify/1$ represents an action with no internal effect on the agent's knowledge base, and that actual notification of the user would require some external program to periodically query the specification. The agent must also send a message acknowledging receipt of every abstract (r_{17}). And since the user will be away from his computer, he decides to forward urgent mail to his temporary new address. This could be enacted by simply stating that urgent messages should be sent to his new address. But he decides to create a new internal action, represented by $forward\,(MsgId, To)$, whose effect is to forward the newly incoming message $MsgId$ to the address To, thus making it easier to specify future forwarding options. The specification of this action is achieved by asserting rule r_{18}. Then, based on this action, he can specify that all urgent messages be forwarded

[3] From now on we omit all propositions and assertions concerning the clock unless relevant for the presentation.

[4] At this state we omit the model and update.

to his new address, by asserting rule r_{19}. Finally, the user realizes that the messages that have been deleted are not being effectively deleted, but rather only removed from their folders, i.e. $msg\,(M, F, S, B, T)$ is still true, except that there is no $in\,(M, \_)$ that is true. He then decides to create another internal action, *purge*, whose effect is that of making false all those messages that have been previously removed from all folders by action *delete*. The specification of this action is obtained via asserting rule r_{20}. The assertion of rules 13–20 constitutes event E_5.

$r_{13} : not\ spam\,(F, S, B) \leftarrow contains\,(F, \text{“accountant”})$
$r_{14} : move\,(M, abstracts) \leftarrow newmsg\,(M, F, S, B)\,, contains\,(S, \text{“abstract”})$
$r_{15} : move\,(M, cfp) \leftarrow newmsg\,(M, F, S, B)\,, contains\,(S, \text{“cfp”})$
$r_{16} : notify\,(M) \leftarrow newmsg\,(M, F, S, B)\,, not\ assert\,(in\,(M, inbox))\,,$
$assert\,(in\,(M, Fldr))$
$r_{17} : send\,(From, S, \text{“Thanks”}) \leftarrow newmsg\,(M, F, S, B)\,, contains\,(S, \text{“abstract”})$
$r_{18} : send\,(To, S, B) \leftarrow forward\,(M, To)\,, newmsg\,(M, F, S, B)$
$r_{19} : forward\,(M, \text{“b@domain”}) \leftarrow newmsg\,(M, \_, \text{“urgent”}, \_)$
$r_{20} : assert\,(not\ msg\,(M, F, S, B, T)) \leftarrow purge, msg\,(M, F, S, B, T)\,, not\ in\,(M, \_)$

At the subsequent update the agent receives more messages, performs a *purge*, moves message 6 to the private folder, and deletes message 6, all encoded by these facts in E_6:

$newmsg(9,\text{‘a2@e’},\text{‘abstract’},\text{‘abs...’}), newmsg(10,\text{‘a3@e’},\text{‘abstract’},\text{‘abs...’})$
$newmsg(13,\text{‘accountant@c’},\text{‘fwd:credit’},\text{‘...’}), newmsg(11,\text{‘x@d’},\text{‘urgent’},\text{‘...’})$
$move(6, inbox, private), delete(6), purge, newmsg(8,\text{‘a1@e’},\text{‘abstract’},\text{‘abs...’})$
$newmsg(12,\text{‘accountant@c’},\text{‘fwd:credit’},\text{‘...’})$

After this event, the stable model contains, on messages 1 and 3, as a result of the *purge*, $assert\,(not\ msg\,(M, F, S, B, T))$; $assert\,(in\,(M, abstracts))$ for messages 8, 9 and 10; plus $forward\,(11, \text{“b@domain”})$ and the corresponding send action, i.e. $send\,(\text{“b@domain”}, \text{“urgent”}, \text{“...”})$; and, concerning message 6, the stable model contains $assert\,(in\,(6, private))$ and $delete\,(6)$. There are also notifications for messages 8, 9, 10 and 14. Next the user decides that whenever a message is both deleted and moved, the deletion action prevails, i.e. it should not be asserted into the folder specified by the move action. This is encoded by the assertion of rule r_{21} (below). Furthermore, the user decides to update his spam rules to avoid all spam his accountant has been forwarding him (r_{22}). Finally, because he wishes the agent to deal with communication with the referees, he sets up the assignments between referees and submitted papers ($r_{23} - r_{28}$). So, E_7 is $\{assert\,(\langle r_{21}\rangle)\,, \ldots, assert\,(\langle r_{28}\rangle)\}$, where:

$r_{21} : not\ assert\,(in\,(M, F_{to})) \leftarrow move\,(M, F_{from}, F_{to})\,, delete\,(M)$
$r_{22} : spam\,(F, S, B) \leftarrow contains\,(S, \text{“credit”})\,, contains\,(S, \text{“Fwd”})$
$r_{23} : assign\,(\text{“paper1”}, \text{“ref2@b”})$ $\quad$ $r_{24} : \text{assign}\,(\text{“paper2”}, \text{“ref2@b”})$
$r_{25} : assign\,(\text{“paper2”}, \text{“ref3@c”})$ $\quad$ $r_{26} : \text{assign}\,(\text{“paper3”}, \text{“ref3@c”})$
$r_{27} : assign\,(\text{“paper3”}, \text{“ref1@a”})$ $\quad$ $r_{28} : \text{assign}\,(\text{“paper1”}, \text{“ref1@a”})$

After all have been asserted, at the subsequent state the agent receives a spam message from the accountant, performs a move and a delete of message

12 to test if the new rule is working, and sends messages to the referees inviting them to review the corresponding papers, as encoded by these facts and rules that belong to E_8:

$$newmsg\,(15, \text{“accountant@c”}, \text{“fwd : credit”}, \text{“...”})\,, move\,(12, inbox, folder1)$$
$$delete\,(12) \qquad send\,(R, PId, \text{“invitation to review”}) \leftarrow assign\,(PId, R)$$

At this point, we invite the reader to check that message 15 was rejected and message 12 was indeed deleted. It is important to notice that messages to referees are sent only once. This is because the rule belonging to E_8 is not an assertion and thus never becomes part of the agent's knowledge base. It is just utilized to determine the stable model at this state, and not used again.

Subsequently the user decides to specify the way the email agent should deal with communication with authors and reviewers. Forthwith, we show how some of these tasks can be specified in EVOLP. Upon receipt of a message from a reviewer accepting to review a given paper, the latter should be sent to the referee the moment it arrives. This can be specified by rule r_{29} (below) asserting a rule that sends the paper to the referee, but this assertion should only take place after the referee accepts the task. If the paper has already been received when the reviewer does so, then it should be sent immediately (r_{30}). Of course, papers received after some deadline, unless some extension was granted to a particular one, should be rejected and the author so notified. This is encoded by rules r_{31} and r_{32} which are asserted when the deadline is reached, even though it might not yet been set. Rule r_{31} sends a message to the author while rule r_{32} prevents the paper being sent to the referee. Finally, the user asserts two rules to deal with deadline extensions on a paper by paper basis. Whenever the user includes an event of the form $dline\,(PId, Dur)$ in an update, he is granting an extension of the deadline to paper PId with duration Dur. This immediately causes $ext\,(PId)$ to be asserted, preventing the paper being rejected. Concurrently, by means of rule r_{34}, a rule is asserted that will render $ext\,(PId)$ false once the deadline plus the extension is reached, after which the paper is rejected. Thus, E_9 is $\{assert\,(\langle r_{29}\rangle)\,, \ldots, assert\,(\langle r_{34}\rangle)\}$, where:

$$r_{29} : assert(send(R,S,B) \leftarrow newmsg(M,F,S,B), contains(S,PId), assign(PId,R)\ ) \leftarrow newmsg(M,R,PId,B), contains(B,\text{‘accept’})$$

$$r_{30} : send(R,PId,B) \leftarrow newmsg(M,R,PId,B_1), contains(B_1,\text{‘accept’}), msg(M_1,F,PId,B,T)$$

$$r_{31} : assert(send(F,S,\text{‘too late’}) \leftarrow newmsg(M,F,S,B), contains(S,PId), not\,ext(PId)\ ) \leftarrow time(T), deadline(T)$$

$$r_{32} : assert(not\,send(Referee,S,B) \leftarrow newmsg(M,F,S,B), contains(S,PId), not\,ext(PId)\ ) \leftarrow time(T), deadline(T)$$

$$r_{33} : assert(ext(PId)) \leftarrow dline(PId,D)$$
$$r_{34} : assert(assert(not\,ext(PId)) \leftarrow time(D+T), deadline(T)) \leftarrow dline(PId,D)$$

Subsequently the user sets the deadline by asserting the fact *deadline* (14)[5], i.e. the event E_{10} contains the fact *assert* (*deadline* (14)).

The remainder of the story goes as follows: at event E_{11} the agent receives both acceptance messages from referee 1; at event E_{12} it receives paper 2; the user grants deadline extensions of two time units to papers 1 and 3, encoded in event E_{13}; at event E_{14} it receives the acceptance messages from referee 2; at event E_{15}, i.e. after the deadline but before the granted extension expires, it receives paper 1; at event E_{16} it receives the acceptance messages from referee 3; at event E_{17}, i.e. after the extension has expired, it receives paper 3. Lack of space prevents us from elaborating further on what folows these events, but we invite the reader to check that: after event E_{14} paper 2 is sent to referee 2; after event E_{15} paper 1 is sent to both referees 1 and 2; after event E_{16} paper 2 is sent to referee 3; since paper 3 arrives after its deadline extension it is rejected, a message being sent to the author, and the paper not sent to any referee.I.e:

$$
\begin{aligned}
E_{11} &= \{newmsg(16, \text{“ref1@a”}, \text{“paper1”}, \text{“accept”}),\\
&\qquad newmsg(17, \text{“ref1@a”}, \text{“paper3”}, \text{“accept”})\}\\
E_{12} &= \{newmsg(18, \text{“a2@e”}, \text{“paper2”}, \text{“thepaper”})\}\\
E_{13} &= \{dline(\text{“paper3”}, 2), \text{dline}(\text{“paper1”}, 2)\}\\
E_{14} &= \{newmsg(19, \text{“ref2@b”}, \text{“paper1”}, \text{“accept”}),\\
&\qquad newmsg(20, \text{“ref2@b”}, \text{“paper2”}, \text{“accept”})\}\\
E_{15} &= \{newmsg(21, \text{“a1@e”}, \text{“paper1”}, \text{“thepaper”})\}\\
E_{16} &= \{newmsg(22, \text{“ref3@c”}, \text{“paper2”}, \text{“accept”}),\\
&\qquad newmsg(23, \text{“ref3@c”}, \text{“paper3”}, \text{“accept”})\}\\
E_{17} &= \{newmsg(24, \text{“a3@e”}, \text{“paper3”}, \text{“thepaper”})\}
\end{aligned}
$$

4 Concluding remarks

A large number of software products is nowadays available to perform email monitoring and filtering. One example is Spam Agent [19], a recently deployed email monitoring and filtering tool which features a comprehensive set of filters (over 1500) to block spam and unwanted emails. Email monitoring and filtering rules can be defined in terms of message sender, recipient, subject, body, and arbitrary combinations of them. The SpreadMsg software email filtering and forwarding agent [12] provides unattended data capture, scanning, and extraction from a wide variety of data. User rule sets are applied to data and, when criteria are met, data are further parsed and turned into messages that can be delivered to email addresses, text pagers, digital cellular or GSM mobile phones, laptops, PDA's etc.. The SuperScout Email Filter [16], besides supporting similar capabilities to the previously mentioned agents, features a Virtual Learning Agent (VLA). The VLA is a content development tool that can be trained to

[5] Dealing with the synchronisation of external and internal times is outside the scope of this paper. Here, we set the deadline as a value that refers to the agent's internal clock.

understand and recognize specific proprietary content in order to protect confidential and business critical information from the security risks arising from accidental or malicious leakage. Widely used commercial email filtering systems provide SPAM filters and, in some cases, a more general set of email handling rules. Outlook, Netscape, and Hotmail, for instance, all provide means to define email filtering rules. Some systems require the user to write the filtering rules, while others employ learning algorithms or try to extract patterns from examples. An interesting, very recent proposal is the Personal Email Assistant (PEA) [5], which is intended to provide a customizable, machine learning based environment to support email processing. An aspect of PEA is that it relies on combining available open source components for information retrieval, machine learning, and agents. Lack of space prevents us from mentioning other (out of many) email monitoring and filtering agents available. It is worth observing however that, to the best of our knowledge, none of the available agents enjoys the ability of autonomously and dynamically updating its own filtering policies in a way as general as the EVOLP specifications illustrated in the present work.

In this paper, we have illustrated how EVOLP can be employed to specify a personal assistant agent for e-mail management on the Web. More generally, we have shown how EVOLP can be employed to express updateable RuleML rule bases (by means of a suitable XML schema — which we did not illustrate for lack of space). This paves the way for a more general usage of EVOLP in the context of the Semantic Web. In this perspective, EVOLP features an implemented inference engine which can be fruitfully used for querying data and for the rapid prototyping of Web-based agents.

The complexity of the inference engine relates to the possible branching of evolutions of a program. Non-stratified rules for assertions can be used to model alternative updates to the agent's knowledge base, e.g. for stating, under certain conditions, either to move a message to a folder or to delete it but not both. Non-stratification can also be used to model uncertainty in the external observations. In both these cases, EVOLP semantics provides several evolution stable models, upon which reasoning can be made, concerning what happens in case one or other action is chosen. On the other hand, by having various models, EVOLP can no longer be used to actually perform the actions, unless some mechanism for selecting models is introduced. For (static) logic programs, this issue of selecting among stable models has already been extensively studied: either by defining more skeptical semantics that always provide a unique model or by preferring among stable models based on some priority ordering on rules. The introduction of such mechanisms in EVOLP too is the subject of current and future work by the authors, with particular emphasis on defining a well-founded based semantics for EVOLP. Another way to view such a well-founded sematics, is that it is sound, though not complete, wrt. the stable model based semantics of Section 2. However, its complexity, rather than being NP-complete as is the stable models semantics, is polynomial on the size of the EVOLP program.

We began this article by extolling the virtue and promise of Logic Programming, for adumbrating the issues and solutions relative to the (internally and

externally) updatable executable specification of agents, and to the study of their evolution by means of a precisely defined declarative semantics with well-defined properties. We have shown, in a concrete web related example, how EVOLP can be used, not only to specify an agent's email policy, but also to dynamically change its specification, and make desired foreseen changes, dependent on dynamic external conditions or events. We have run and tested the example, with an EVOLP implementation available at: http://centria.di.fct.unl.pt/~jja/updates/.

Acknowledgments

This work has been partially supported by project FLUX "Flexible Logical Updates", POSI/SRI/40958/2001, financed by FEDER.

References

1. The Semantic Web Activity. `http://www.w3.org/2001/sw/`.
2. J. J. Alferes, A. Brogi, J. A. Leite, and L. M. Pereira. Evolving logic programs. In *JELIA'02*, volume 2424 of *LNAI*. Springer, 2002.
3. J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *Journal of Logic Programming*, 45(1-3), 2000.
4. J. J. Alferes, L. M. Pereira, H. Przymusinska, and T. Przymusinski. LUPS : A language for updating logic programs. *Artificial Intelligence*, 138(1-2), 2002.
5. R. Bergman, M. Griss, and C. Staelin. A personal email assistant. Technical Report HPL-2002-236, HP Labs Palo Alto, 2002.
6. H. Boley, S. Tabet, and G. Wagner. Design rationale of ruleml: A markup language for semantic web rules. In *SWWS'01*, 2001.
7. The World Wide Web Consortium. `http://www.w3.org/`.
8. T. Eiter, M. Fink, G. Sabbatini, and H Tompits. A framework for declarative update specifications in logic programs. In *IJCAI'01*. Morgan-Kaufmann, 2001.
9. M. Gelfond and V. Lifschitz. The stable semantics for logic programs. In *ICLP'88*. MIT Press, 1988.
10. B. Grosof. Representing e-business rules for the semantic web: Situated courteous logic programs in ruleml. In *WITS'01*, 2001.
11. B. Grosof and T. Poon. Representing agent contracts with exceptions using xml rules, ontologies, and process descriptions. In *RuleML-BR-SW'02*, 2002.
12. Compuquest Inc. Spreadmsg. www.compuquestinc.com.
13. The Rule Markup Initiative. `http://www.dfki.uni-kl.de/ruleml/`.
14. J. A. Leite. *Evolving Knowledge Bases*. IOS Press, 2003.
15. V. Lifschitz and T. Woo. Answer sets in general non-monotonic reasoning (preliminary report). In *KR'92*. Morgan-Kaufmann, 1992.
16. Caudex Services Ltd. Superscout email filter. www.caudexservices.co.uk.
17. Resource Description Framework (RDF). `http://www.w3.org/RDF/`.
18. M. Schroeder and G. Wagner, editors. *Procs. of RuleML-BR-SW'02*, number 60 in CEUR-WS Publication, 2002.
19. Spam-Filtering-Software.com. Spam agent. www.spam-filtering-software.com.
20. Extensible Markup Language (XML). `http://www.w3.org/XML/`.