

Logic Programming Updates

José Júlio Alves Alferes

Logic Programming for KRR

- Logic Programming has been widely and successfully used for Knowledge Representation and Reasoning
 - It is a logic language with a precise declarative semantics allowing for formal representation of knowledge
 - It comes with derivation procedures, and corresponding implementations, allowing for reasoning
 - It is a non-monotonic logic, allowing for dealing with incomplete, common sense, knowledge
 - It is a rule language (arguably) easy to use and reason with (e.g. for modelling business rules)

LP and Knowledge Evolution

- Logic Programming is non-monotonic
 - It allows for the incorporation of new knowledge to change previous conclusions
 - If new knowledge contradicts prior assumptions, LP is able to accommodate it by non-monotonically revising (default) assumptions
- This is appropriate for dealing with new knowledge that further completes what is known about the (static) world
 - Typical case of *Belief Revision*

LP and Knowledge Updates

- What if the world itself is changing?
 - New knowledge may contradict previous factual (certain) knowledge
 - In this case revising assumptions is not enough
- Logic Programming lacks means to
 - Integrate knowledge updates (from external sources)
 - Deal with external events
 - Represent rules about the transition between world states

Reasoning about changes

- Other (non-monotonic) languages and methodologies have been defined to deal with changes in the world
 - Situation calculus, event calculus, fluent calculus, action languages, ...
- In these, it is possible to deal with external events, and with knowledge updates
- It is also possible to specify rules about transitions that describe ways in which the world changes

Rule Updates

- These languages and methodologies are basically concerned with facts that change
 - There is a set of fluents (facts)
 - There are static rules describing the domain, which are themselves not subject to change
 - There are dynamic rules describing how the facts may change due to actions (also not subject to change)
- What if the rules themselves (static or dynamic) change?
 - In general, the rules of a given domain may change in time
 - Even rules that describe the effects of actions (behaviour) may change in practical applications
- In this context, previous languages don't help.

Languages for Rule Updates

- Languages dealing with *highly dynamic environments* where, besides facts, also static and dynamic rules may change
- Such languages need means for:
 - Incorporating knowledge updates from external sources
 - changes in rules coming from “user”
 - events from the environment
 - Describing rules about transitions between states
 - Describing self-updates of the knowledge base, and combining these with external ones

Summary

1. Define the meaning of updating a (evolving) logic programming knowledge base with new rules
 - I.e. define the meaning of updating a program by other programs
2. Extend Logic Programming to allow for updates and for dynamic rules
3. Illustrate the language in application domains
4. Mention further extensions and results

Dynamic Logic Programming

Dealing with sequences of LPs

- Dynamic Logic Programming was introduced to deal with updates to logic programs
 - It gives meaning to sequences of LPs
- Intuitively a sequence of LPs is the result of updating P_1 with the rules in P_2, \dots
 - But different programs may also come from different hierarchical instances, viewpoints (with preferences), etc
- Inertia is applied to rules rather than to literals
 - Older rules conflicting with newer applicable rules are rejected

Updating Models isn't enough

- When updating logic programs doing it model by model is not desired
 - It loses the directional information of the logic programming arrow

P_1 : $\text{sleep} \leftarrow \text{not tv_on}$ $M = \{\text{tv}, w\}$
 $\text{watch} \leftarrow \text{tv_on}$
 tv_on

P_2 : $\text{not tv_on} \leftarrow \text{p_failure}$ $M = \{\text{pf}, w\}$
 p_failure

P_3 : not p_failure $M = \{w\}$

Updating Models isn't enough

- When updating logic programs doing it model by model is not desired
 - It loses the directional information of the logic programming arrow

P₁: sleep ← not tv_on
watch ← tv_on
tv_on

M = {tv, w}

P₂: not tv_on ← p_failure
p_failure

M = ~~{pf, w}~~ {pf, s}

P₃: not p_failure

M = ~~{w}~~ {tv, w}

- Inertia should be applied to rules rather than to the previous consequences

Incorporation of new rules

- One should not have to worry about how to incorporate new rules

- The semantics should take care of it

P_1 : $\text{auth}(\text{Person}) \leftarrow \text{employee}(\text{Person})$
 $\text{not auth}(\text{Person}) \leftarrow \text{on_leave}(\text{Person})$

P_2 : $\text{auth}(\text{Person}) \leftarrow \text{director}(\text{Person})$

P_3 : $\text{not auth}(\text{Person}) \leftarrow \text{max_sec}, \text{not safety}(\text{Person})$

- Instead of rewriting the program, we should simply update it with the new rules
- The semantics should consider the last update, plus all rules of the previous that don't conflict

Dynamic Logic Programs

- A *dynamic logic program* \mathcal{P} is a sequence of generalised LPs (i.e. programs possibly with negation in rule heads):

$$\mathcal{P} = (P_1, P_2, \dots, P_n)$$

- An interpretation M is a *stable model of \mathcal{P} at state n* iff:

$$M = \text{least}([\rho(\mathcal{P}) - \text{Rej}_s(\mathcal{P}, M)] \cup \text{Def}_s(\mathcal{P}, M))$$

where $\rho(\mathcal{P})$ is the multi-set of all rules in \mathcal{P}

Rejection and Defaults

- By default assume the negation of atoms that have no rule for it with true body (given M)

$$\text{Def}_s(\mathcal{P}, M) = \{\text{not } A \mid \nexists r \in P_i \ i \leq s, H(r) = A, M \models B(r)\}$$

- Reject all rules with head L that belong to a former program, if there is a later one with complementary head and a true body

$$\text{Rej}_s(\mathcal{P}, M) = \{r \in P_j \mid \exists r' \in P_i, j \leq i \leq s, \\ H(r) = \text{not } H(r'), M \models B(r')\}$$

Relation to extant

- Generalises Answer-Sets
 - If there is a single program in the sequence, then coincides with Answer-Set Semantics
 - Even with sequences, just differs from Answer-Sets when there are conflicts
 - Without conflicts, it coincides with the answer-sets of the union
- Generalises Interpretation Updates
 - Just update the interpretations (as facts) with the new program

Further developments

- Further properties, principles, and relation to other approaches
- Account of rejection based on level-mappings
 - Similar to some form of stratification
- Operational Semantics for Dynamic Logic Programs
 - With corresponding implementation resorting to state-of-the-art Answer-Set Solvers

Further developments (2)

- Weaker (less expressive, and with lower computational complexity) semantics generalising Well-Founded Semantics
 - Also with corresponding implementation based on XSB-Prolog
- Relation and combination with preferences among rules
- Multi-dimensional Logic Programs
 - Considers structures richer than sequences
 - Programs are arranged as any partial order
 - Allows several dimensions of rejection
 - Time updates, hierarchical relations, viewpoints,...

Evolving Logic Programs

What is still missing

- Dynamic Logic Programs give meaning to sequences
- But how to come up with those sequences?
 - Changes may be additions or retractions
 - Updates may be conditional on a present state
 - Some rules may represent (persistent) laws
- Since LP can be used to describe knowledge states and also sequences of updating states, it's only fit that LP is used too to describe transitions, and thus create such sequences

LP Update Languages

- Extend logic programming with features that allow to define dynamic (state transition) rules
 - Add, on top of Logic Programming, a language with meaningful commands that generate sequences (LUPS, EPI, KABUL) or
 - ▶ Extend the basic Logic Programming language minimally in order to allow for this generation of sequences (EVOLP)

What is needed to evolve

- For allowing for program evolution
 - Meaning of (evolving) programs should be sequences of sets of literals, each representing a state
 - A construct to assert new information is needed
 - nots in head are needed to allow newer to supervene older rules
- Program evolution may be influenced from outside
 - Allow for external events
 - ... ideally written in the same language as programs

EVOLP Syntax

- EVOLP rules are generalised Logic Program rules plus special predicate `assert/1`
- The argument of `assert` is an EVOLP rule (i.e. arbitrary nesting of `assert` is allowed)
- Examples:
 `assert(a ← not b) ← d, not e`
 `not a ← not assert(assert(a ← b) ← not b), c`
- EVOLP programs are sets of EVOLP rules

Meaning of Self-evolving LPs

- Determined by sequences of sets of literals
- Each sequence represents a possible evolution
- The n^{th} set in a sequence represents what is true / false after n steps in that evolution
- The first set in sequences is an answer-set of the program, where `assert / 1` literals are viewed as normal ones
- If `assert(Rule)` belongs to the n^{th} set then $(n+1)^{\text{th}}$ sets must consider the addition of Rule

Intuition in example

$a \leftarrow$
 $\text{assert}(b \leftarrow)$
 $\text{assert}(c \leftarrow) \leftarrow b$

- At the beginning a is true, and so is $\text{assert}(b \leftarrow)$
- Therefore, rule $b \leftarrow$ is asserted
- At 2nd step, b becomes true, and so does $\text{assert}(c \leftarrow)$
- Therefore, rule $c \leftarrow$ is asserted
- At 3rd step, c becomes true.

($\{a, \text{assert}(b \leftarrow)\}$, $\{a, b, \text{assert}(b \leftarrow), \text{assert}(c \leftarrow)\}$,
 $\{a, b, c, \text{assert}(b \leftarrow), \text{assert}(c \leftarrow)\}$)

Self-evolution definitions

- An *evolution interpretation* of P over \mathcal{L} is a sequence (I_1, \dots, I_n) of sets of atoms from \mathcal{L}_{as}
- The *evolution trace* of (I_1, \dots, I_n) given P is (P_1, \dots, P_n) where

$$P_1 = P \text{ and } P_i = \{R \mid \text{assert}(R) \in I_{i-1}\} \quad (2 \leq i \leq n)$$

- Evolution traces contain the programs imposed by interpretations
- We have now to check whether each n^{th} set complies with the programs up to $n-1$

Evolution Stable Models

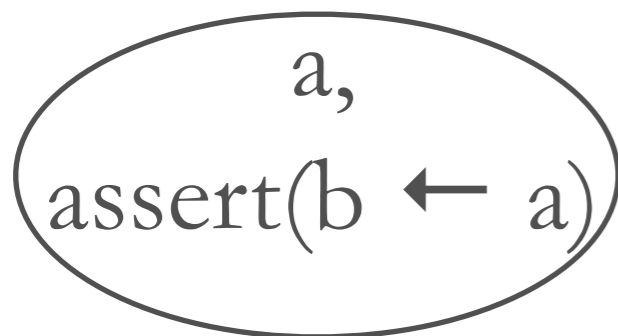
- Given an evolution interpretation and a program, the trace is a dynamic logic program
- Evolution interpretations that, at each state i , have a set which is a stable model of that dynamic logic program at i , are called evolution stable models
- (I_1, \dots, I_n) is an *evolution stable model* of P iff each $I_i, 1 \leq i \leq n$, is a stable model at state i of its trace given P

Simple example

- $(\{a, \text{assert}(b \leftarrow a)\}, \{a, b, c, \text{assert}(\text{not } a \leftarrow)\}, \{\text{assert}(b \leftarrow a)\})$ is an evolution SM of P:

$a \leftarrow$ $\text{assert}(\text{not } a \leftarrow) \leftarrow b$
 $\text{assert}(b \leftarrow a) \leftarrow \text{not } c$ $c \leftarrow \text{assert}(\text{not } a \leftarrow)$

- The trace is $(P, \{b \leftarrow a\}, \{\text{not } a \leftarrow\})$

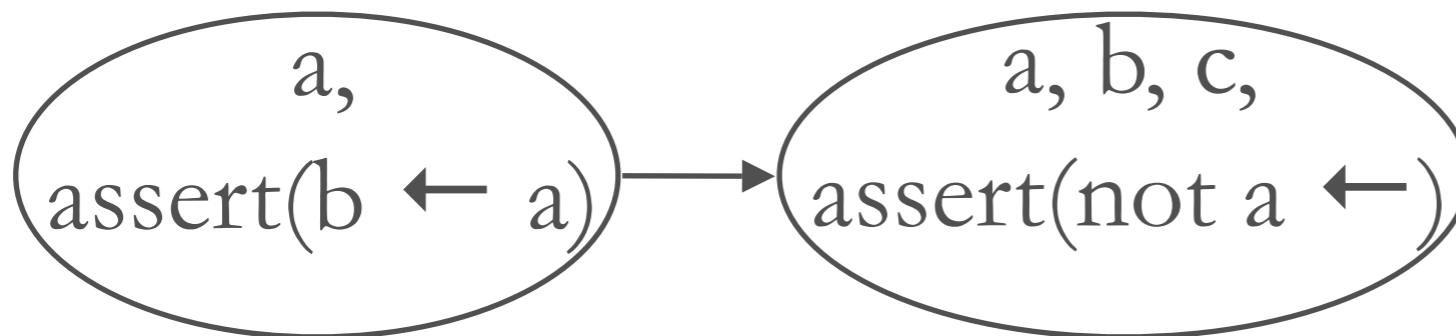


Simple example

- $(\{a, \text{assert}(b \leftarrow a)\}, \{a, b, c, \text{assert}(\text{not } a \leftarrow)\}, \{\text{assert}(b \leftarrow a)\})$ is an evolution SM of P :

$a \leftarrow$ $\text{assert}(\text{not } a \leftarrow) \leftarrow b$
 $\text{assert}(b \leftarrow a) \leftarrow \text{not } c$ $c \leftarrow \text{assert}(\text{not } a \leftarrow)$

- The trace is $(P, \{b \leftarrow a\}, \{\text{not } a \leftarrow\})$

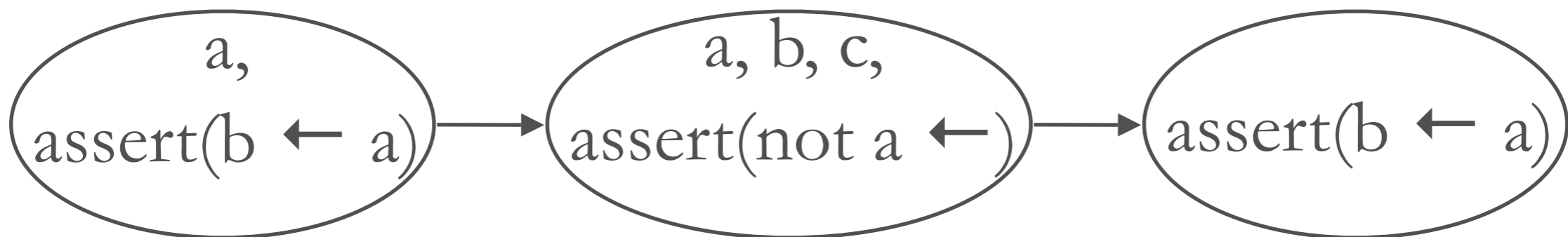


Simple example

- $(\{a, \text{assert}(b \leftarrow a)\}, \{a, b, c, \text{assert}(\text{not } a \leftarrow)\}, \{\text{assert}(b \leftarrow a)\})$ is an evolution SM of P:

$a \leftarrow$ $\text{assert}(\text{not } a \leftarrow) \leftarrow b$
 $\text{assert}(b \leftarrow a) \leftarrow \text{not } c$ $c \leftarrow \text{assert}(\text{not } a \leftarrow)$

- The trace is $(P, \{b \leftarrow a\}, \{\text{not } a \leftarrow\})$

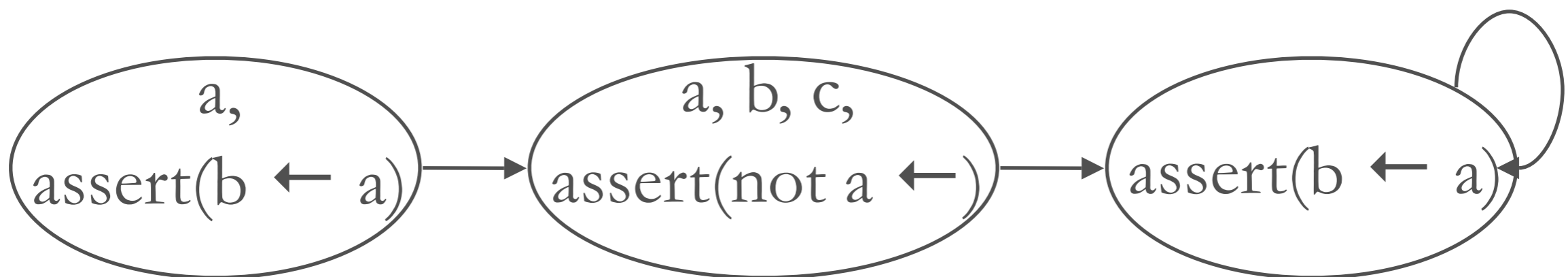


Simple example

- $(\{a, \text{assert}(b \leftarrow a)\}, \{a, b, c, \text{assert}(\text{not } a \leftarrow)\}, \{\text{assert}(b \leftarrow a)\})$ is an evolution SM of P:

$a \leftarrow$ $\text{assert}(\text{not } a \leftarrow) \leftarrow b$
 $\text{assert}(b \leftarrow a) \leftarrow \text{not } c$ $c \leftarrow \text{assert}(\text{not } a \leftarrow)$

- The trace is $(P, \{b \leftarrow a\}, \{\text{not } a \leftarrow\})$

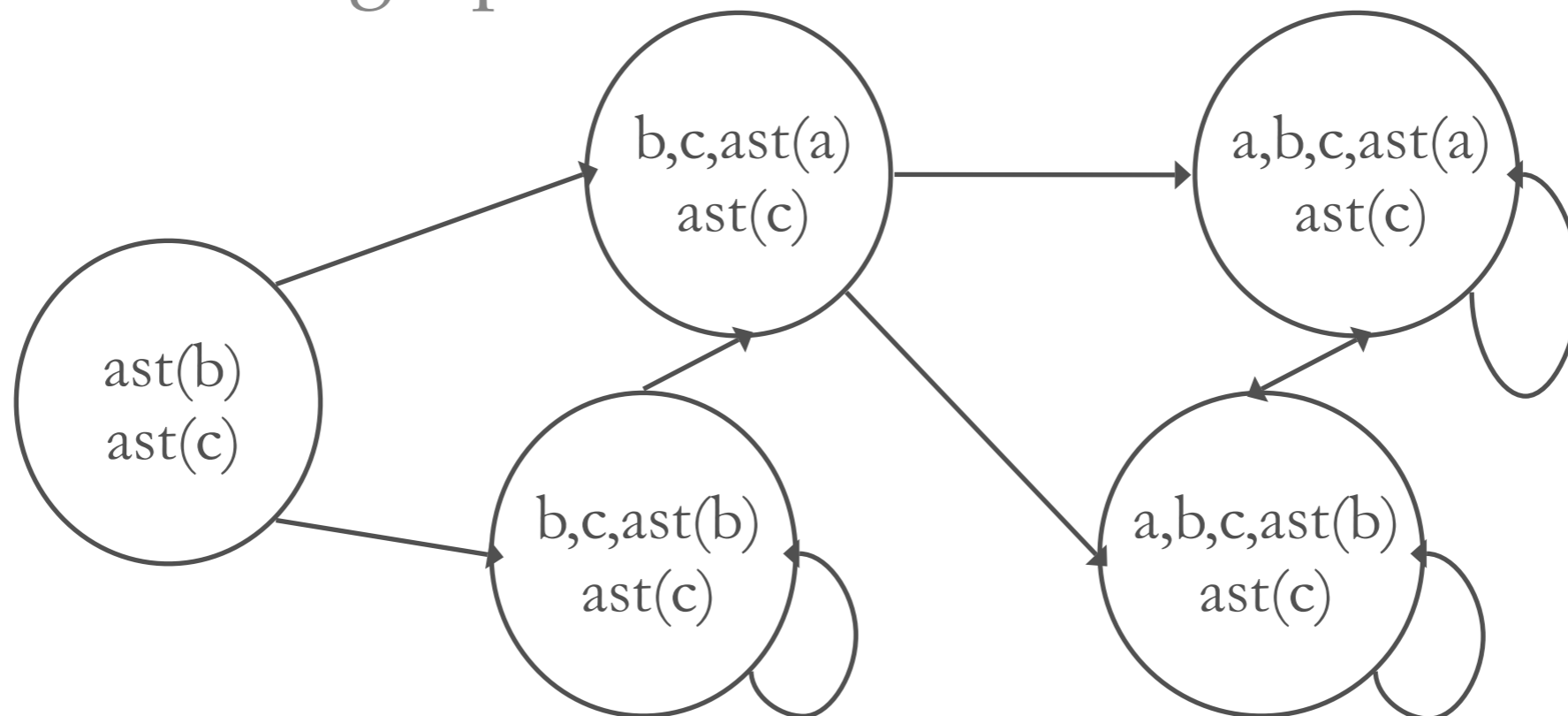


Several evolutions example

- No matter what, assert c; if a is not going to be asserted, then assert b; if c is true, and b is not going to be asserted, then assert a.

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

- Paths in the graph below are evolution SMs



Event-aware programs

- Self-evolving programs are autistic!
- Events may come from the outside:
 - Observations of facts or rules
 - Assertion orders
- Both can be written in the EVOLP language
- Influence from outside should not persist by inertia

Event-aware programs

- An *event sequence* (E_1, \dots, E_n) is a sequence of sets of EVOLP rules
- Given program P and event sequence (E_1, \dots, E_n) , the evolution interpretation (I_1, \dots, I_n) is an *evolution stable model* iff

$\forall_{1 \leq i \leq n}, I_i$ is a stable model of $(P_1, \dots, P_{i-1}, P_i \cup E_i)$

- Note that events are only added to the last state
- This way events do not persist by inertia

Incremental construction

- Evolution stable models can be incrementally constructed
- Operator $\mathcal{T}_{\mathcal{P}}$ determines a set of evolution interpretation of length $n+1$, given another set of length n , and a set of events

$$\mathcal{T}_{\mathcal{P}}(I, E) = \{(I_1, \dots, I_i, I_{i+1}) \mid (I_1, \dots, I_i) \in I \text{ and } I_{i+1} \text{ is a stable model of } (P_1, \dots, P_i, P_{i+1} \cup E)\}$$

- I is an evolution SM of P given (E_1, \dots, E_n) iff I belong to \mathcal{I}_n in the sequence:
 - $\mathcal{I}_1 = \{ (I) \mid I \text{ is a stable model of } P \cup E_1 \}$
 - $\mathcal{I}_{i+1} = \mathcal{T}_{\mathcal{P}}(\mathcal{I}_i, E_{i+1})$

Relation to extant

- EVOLP extends the syntax and semantics of logic programs
 - If no events are given, and no asserts are used, the semantics coincides with answer-sets
- EVOLP was show to properly embed action languages \mathcal{A} , \mathcal{B} , and \mathcal{C} .
 - But further allowing for updates of rules and of behaviour

Relation to extant (2)

- All features of other LP update languages are EVOLP features:
 - Rule updates; Persistent updates; simultaneous updates; events; commands dependent on other commands; ...
- Many extra features can be *programmed*:
 - Updates of update-rules (already in KABUL)
 - Commands that span several states
 - Events with incomplete knowledge
 - Updates of persistent laws
 - Assignments
 - ...

Example applications

Reasoning about actions

- There is a turkey, initially alive:
`alive(turkey)`
- Whenever you shoot with a loaded gun, the turkey at which you shoots dies, and the gun becomes unloaded
`assert(not alive(turkey)) ← loaded, shoot.`
`assert(not loaded) ← shoot.`
- Loading a gun results in a loaded gun
`assert(loaded) ← load.`
- Events of shoot, load, wait, etc make the program evolve
- After some time, the shooter becomes older, has sight problems, and does no longer hit the turkey if without glasses. Add event:
`assert(not assert(not alive(turkey))) ← not glasses)`

Laws and regulations

- A driver loses her license after a 2nd fine. She regains the licence after undergoing a refresher course
 - assert(not license(X) \leftarrow fined(X)) \leftarrow fined(X)
 - assert(license(X)) \leftarrow attendSchool(X)
 - assert(not fined(X)) \leftarrow attendSchool(X).
- Once Republicans take over Congress and Presidency they punish abortion with jail; once Democrats take over both, they don't punish abortion
 - assert(jail(X) \leftarrow abortion(X)) \leftarrow repCongress, repPres
 - assert(not jail(X) \leftarrow abortion(X)) \leftarrow not repCongress, not repPres
- A proposed law *rule* is in place after first being voted by the parliament and then approved by the president
 - assert(assert(*law*) \leftarrow approved(*law*)) \leftarrow voted(*law*)

Policies changes

- An account accepts deposits and withdrawals. The latter are only possible when there is enough balance:
assert(balance(Ac,B+C)) ← changeB(Ac,C), balance(Ac,B)
assert(not balance(Ac,B)) ← changeB(Ac,C), balance(Ac,B)
changeB(Ac,D) ← deposit(Ac,D)
changeB(Ac,-W) ← withdraw(Ac,W), balance(Ac,B), $B > W$.
- The bank now changes its policy, and no longer accepts withdrawals under 50 €:
assert(not changeB(Ac,D) ← deposit(Ac,D), $D < 50$)
- Next VIP accounts are allowed negative balance up to account specified limit:
assert(changeB(Ac,-W) ← vip(Ac,L), withdrawl(Ac,W), $B+L > W$)

Email agent example

- Personal assistant agent for e-mail management able to:
 - Perform basic actions of sending, receiving, deleting messages
 - Storing and moving messages between folders
 - Filtering spam messages
 - Sending automatic replies and forwarding
 - Notifying the user of special situations
- All of this may depend on user specified criteria
- The specification may change dynamically
 - and we don't expect the user to manually change (and debug) previous policies.

Simple rules for email

- By default messages are stored in the inbox:
assert(msg(M,F,S,B,T)) ←
 newmsg(M,F,S,B), time(T), not delete(M)
assert(in(M,inbox)) ← newmsg(M,F,S,B), not delete(M)
assert(not in(M,F)) ← delete(M), in(M,F).
- Spam messages are to be deleted
delete(M) ← newmsg(M,F,S,B), spam(F,S,B)
- The definition of spam can be done by LP rules
spam(F,S,B) ← contains(S,credit)
- This definition can later be updated
not spam(F,S,B) ← contains(F,my_accountant)

Some more rules

- Messages can be automatically moved to other folders. When that happens (not shown here) the user wants to be notified:
 $\text{notify}(M) \leftarrow \text{newmsg}(M,F,S,B), \text{assert}(\text{in}(M,F)), \text{assert}(\text{not in}(M,\text{inbox}))$.
- When a message is marked both for deletion and automatic move to another folder, the deletion should prevail
 $\text{not assert}(\text{in}(M,F)) \leftarrow \text{move}(M,F), \text{delete}(M)$
- The user is organizing a conference, assigning papers to referees. After receipt of a referee's acceptance, a new rule is to be added, which forwards to the referee any messages about assigned papers
 $\text{assert}(\text{send}(R,S,B1) \leftarrow \text{newmsg}(M1,F,S,B1), \text{contains}(S,\text{Id}), \text{assign}(\text{Id},R))$
 $\leftarrow \text{newmsg}(M2,R,\text{Id},B2), \text{contains}(B2,\text{accept})$.

Further developments

Reactive Rules

- Reactive rules with Update Languages
 - Event-Condition-Action rules
 - With event-algebras
 - Though it is possible to express complex events in EVOLP, it results much more natural to have them expressed by algebras
 - Process algebras for actions
- Comparisons to extant work in reactive rules

Execution of reactive rules

- Having several evolution paths is desirable for reasoning about evolutions
 - Specially in the presence of incomplete knowledge
- But for execution it would be desirable to have a single path (and possibly less complexity)
 - Definition of a weaker semantics, with lower complexity, and guaranteeing a single evolution
- Paraconsistency is also an issue for execution, that deserved further developments

Multi-dimensions Updates

- As with dynamic logic programs, also update languages can act in several dimensions (time, hierarchical relation, ...)
 - Update languages building partial orders rather than graphs
 - Used in Multi-Agent Systems for dealing with hierarchies of agent that evolve in time
- Basis for LP Update languages-based agent architectures

Abduction and planning

- Deductive reasoning over update programs allows to deal with evolving knowledge, and know what is true at each state
- Abductive reasoning in this setting amounts to determine what needs to occur in order to reach some (desired) state
 - Similar to planning, looking at future states
- Further developments includes studies of abduction (together with preferences) in the context of update languages

Temporal Operators

- Update language only directly allow to state conditions on the current state
 - What about conditions on previous states, and conditions that span over several states?
- Addition of temporal LTL-like modal operators
 - and embedding of these into EVOLP
- Allows for comparisons to temporal logics using such operators
 - monotonic basis of update languages
 - relation to e.g. temporal situation calculus

Implementations

- They exist:
 - For EVOLP
 - For weaker, well-founded based, semantics
 - Integrated in a general framework for reactive rules in the Web
- The integration in the Semantic Web setting raises the problem of what to do with (monotonic) ontology knowledge

Rules and Ontologies

- How to combine
 - non-monotonic rules of logic programming and update languages **with**
 - monotonic ontological knowledge?
- And don't ontologies evolves?
- These are topics of current research, that are expected to produce results soon...

Conclusions

- Logic Programming Update Languages are a powerful yet simple proposition for representing knowledge in highly dynamic environments
- They form a firm formal basis in which to express, implement, and reason about dynamic knowledge bases
- They have been used in several application areas (reasoning about actions, legal reasoning, MASs, policies specification, Web)
- Some of these areas raise new topics for research...