Modules and Signature Declarations for A-Prolog: Progress Report

Marcello Balduccini
Knowledge Representation Lab
Computer Science Department
Texas Tech University

May 14, 2007

Current Situation

- A-Prolog lacks well-established software engineering tools and methodologies to help in encoding knowledge about complex domains.
- Most existing proposals involve a substantial language re-design, or a drastic shift of perspective (e.g. adoption of object oriented programming paradigm).

As a result, most people still use “basic” A-Prolog, and come up with ad-hoc solutions for the development and integration of complex programs.

Our Approach

We propose a *small* extension of A-Prolog, called *RSig*, that:

- Does not involve any shift in perspective.
- Can be learned easily.
- Involves only minor changes to existing A-Prolog programs.
- Can be easily implemented on top of existing grounding software (an extension of lparse is available online).

We believe this is an important step in bridging the gap between current program development and the sophisticated languages that have been proposed.

A Motivating Example

Let us build a small theory of chemical weapons, Π_w:

$nerve\_agent(tabun).$ $nerve\_agent(sarin).$
$choking\_agent(phosgene).$ $choking\_agent(chlorine).$

% Ontology of agents
$agent(A) \leftarrow nerve\_agent(A).$
$agent(A) \leftarrow choking\_agent(A).$

% Agents are normally deadly
$deadly(A) \leftarrow agent(A), \text{not } ab(A), \text{not } \neg deadly(A).$

% Choking agents are not deadly (in low-to-medium dosage)
$\neg deadly(A) \leftarrow choking\_agent(A).$

% If a deadly agent has been employed, order evacuation
$o(order(evacuation), T) \leftarrow agent(A), h(employed(A), T), deadly(A).$

A Motivating Example (cont'd)

Now consider a program to monitor intelligence agent reliability, Π_m, from [Gianoutsos, 2005]:

% Normally, agent's reports are true.
$h(L,T) \leftarrow about\_step(R,T), content(R,L),$ not $ab(R,T).$
% An exception to this are currently unemployed agents.
$ab(R,T) \leftarrow agent(A), author(R,A), h(\neg employed(A),T).$

Let S_1 be the scenario:

$report(r_1).\ \ about\_step(r_1, 0).\ \ author(r_1, john).\ \ content(r_1, no\_danger).$
$agent(john).\ \ h(employed(john), 0).$

As one would expect, $S_1 \cup \Pi_m$ entails $h(no\_danger, 0)$.

Chemical weapons are not involved, so $S_1 \cup \Pi_m$, Π_w are unrelated.

However, $S_1 \cup \Pi_m \cup \Pi_w$ *also entails* $o(order(evacuation), 0)$*!!*

A Motivating Example (cont'd)

$o(order(evacuation), 0)$ follows from:

- $agent(john). \quad h(employed(john), 0).$
- $deadly(A) \leftarrow agent(A), \text{not } ab(A), \text{not } \neg deadly(A).$
- $o(order(evacuation), T) \leftarrow agent(A), h(employed(A), T), deadly(A).$

There is unintended interaction between $S_1 \cup \Pi_m$ and Π_w, because $agent$ (and $employed$) have different meanings in the two programs.

In practice, this substantially complicates the development of large programs.

Proposed Solution

- Structure programs in *modules*.
- Each module has a clearly defined input/output interface.
- Internal relations and functions (not part of the interface) are hidden from the other modules.
- The signature of the language used in each module is explicitly specified.

Motivating Example, Revisited

Consider a module M_w containing Π_w and input/output interface:

$\#module$ chem_agents.
$\#import\ rel\ h(\_,\_).$
$\#export\ rel\ agent(\_).$
$\#export\ rel\ o(\_,\_).$
$\vdots$

and a module M_m for Π_m, with interface:

$\#module$ int_monitor.
$\#import\ rel\ h(\_,\_).$
$\#import\ rel\ agent(\_).$
$\#import\ rel\ about\_step(\_,\_), author(\_,\_), \ldots.$
$\vdots$

From the declarations, the interaction between the two modules is now evident.

Removing the Interaction

Let us modify the interface of M_w as follows:

$\#module$ chem_agents.
$\#import\ rel\ h(\_,\_).$
$\#export\ rel\ chem\_agent(\_).$
$\#export\ rel\ o(\_,\_).$
$\vdots$

and add to the module a rule:

$$chem\_agent(A) \leftarrow agent(A).$$

which maps the input/output relation $chem\_agent$ to the internal relation $agent$.

$S_1 \cup M_m \cup M_w$ *does not entail* $o(order(evacuation), 0)$.

Specifying the Signature

In *RSig*, the signature of a relation r is specified by a statement:

$$\#sig\ rel\ r(p_1, p_2, \ldots, p_k).$$

where p_i's are names of sorts.
Informal reading: "*the arguments of r are of sorts p_1, ..., p_k.*"

The signature of a function f is specified as:

$$\#sig\ func\ f(p_1, p_2, \ldots, p_k) \rightarrow p_0.$$

Informal reading: "*the arguments of f are of sorts p_1, ..., p_k, and the terms formed by f are of sort p_0.*"

Signatures for the Example

Signature of the input for the example:

$\#sig\ rel\ about\_step(report, time),\ author(report, agent),\ content(report, fluent).$
$\#sig\ rel\ h(fluent, time),\ o(action, time).$
$\#sig\ func\ employed(agent) \rightarrow fluent,\ no\_danger \rightarrow fluent.$

together with sort definitions, e.g.:

$$time(0).\ time(1).\ time(2).\ \ldots$$
$$report(r_1).\ report(r_2).\ \ldots$$

Signature for M_w:

$$\#sig\ rel\ deadly(agent), ab(agent).$$

Signature for M_m:

$$\#sig\ rel\ ab(report, time).$$

Advantages of Signatures

- Improved readability of the program.
- Compared with lparse-style grounding:
  - ◇ Free use of variables – no need to remember and comply with association variable $\leftrightarrow$ domain.
  - ◇ Simplified handling of cardinality atoms.
- Compared with dlv-style grounding: smaller ground instance.

Cardinality Atoms: lparse and RSig

Consider the following lparse programs and their groundings:

$$P_1 = \left\{ \begin{array}{l} d(0..2). \quad \#domain\ d(Y). \\ 1\{p(YY) : d(YY)\}1. \end{array} \right. \qquad gr(P_1) = \left\{ \begin{array}{l} 1\{p(0), p(1), p(2)\}1. \end{array} \right.$$

$$P_2 = \left\{ \begin{array}{l} d(0..2). \quad \#domain\ d(Y). \\ 1\{p(Y) : d(Y)\}1. \end{array} \right. \qquad gr(P_2) = \left\{ \begin{array}{l} 1\{p(0)\}1.\ 1\{p(1)\}1. \\ 1\{p(2)\}1. \end{array} \right.$$

In *RSig*, the programs:

$$P_3 = \left\{ \begin{array}{l} d(0..2). \\ \#sig\ rel\ p(d). \\ 1\{p(Y)\}1. \end{array} \right. \qquad P_4 = \left\{ \begin{array}{l} d(0..2). \\ \#sig\ rel\ p(d). \\ 1\{p(YY)\}1. \end{array} \right.$$

have the same grounding:

$$1\{p(0), p(1), p(2)\}1.$$

Conclusions

- Most existing software engineering methodologies involve substantial language redesign, or shift of perspective (e.g. OOP).
- This discourages their adoption, in particular for existing programs/projects.

◇ *RSig* is a *small* extension of A-Prolog.

◇ It does not involve any shift of perspective or language redesign.

◇ Can be learned easily.

◇ Involves only minor changes to existing programs.

◇ Can be easily implemented on top of existing grounding software (extention of lparse available online).

The Complete Program

$\#sig\ rel\ about\_step(report, time),\ author(report, agent),\ content(report, fluent).$
$\#sig\ rel\ h(fluent, time),\ o(action, time).$
$\#sig\ func\ employed(agent) \rightarrow fluent,\ no\_danger \rightarrow fluent.$

$\#module$ chem_agents.
$\#import\ rel\ h(\_, \_).$
$\#export\ rel\ chem\_agent(\_),\ o(\_, \_).$
$\#sig\ rel\ deadly(agent), ab(agent).$

$chem\_agent(A) \leftarrow agent(A).$
$nerve\_agent(tabun).\ \ \ldots.$
$agent(A) \leftarrow nerve\_agent(A).\quad agent(A) \leftarrow choking\_agent(A).$
$deadly(A) \leftarrow agent(A), \text{not } ab(A), \text{not } \neg deadly(A).$
$\neg deadly(A) \leftarrow choking\_agent(A).$
$o(order(evacuation), T) \leftarrow agent(A), h(employed(A), T), deadly(A).$

$\#endmodule.$

$\#module$ int_monitor.
$\#import\ rel\ h(\_, \_), agent(\_), about\_step(\_, \_), author(\_, \_), \ldots.$
$\#sig\ rel\ ab(report, time).$

$h(L, T) \leftarrow about\_step(R, T), content(R, L), \text{not } ab(R, T).$
$ab(R, T) \leftarrow agent(A), author(R, A), h(\neg employed(A), T).$

$\#endmodule.$