Modules and Signature Declarations for A-Prolog: Progress Report

Marcello Balduccini

Computer Science Department
Texas Tech University
Lubbock, TX 79409 USA
marcello.balduccini@ttu.edu

Abstract. It has been demonstrated that A-Prolog can be used effectively to encode knowledge about complex domains. However, there is still a lack of well-established software engineering inspired tools and methodologies aimed at helping the programmer in this task. Rather than going through a substantial redesign of the language, as in most approaches from the literature, our purpose here is to propose a *light-weight* extension of the language, introducing only a few simple constructs with straightforward semantics, and nonetheless providing key support for simple modular design of programs. Drawing from our experience of encoding knowledge in A-Prolog, we identify two main requirements that, we believe, need to be satisfied by such a simple extension of A-Prolog. Next, we design our extension of A-Prolog, called *RSig* to satisfy these requirements. A parser for *RSig* has been implemented, based on LPARSE, and is available online. It is our belief that *RSig* can be quickly learned and used by average A-Prolog users to both write new programs and restructure existing programs. We also hope that the experience with *RSig* can promote the transition towards more sophisticated extensions of A-Prolog.

1 Introduction

As demonstrated by several authors in recent years (see for example [18, 17, 8, 3]), A-Prolog [10, 11] is a powerful knowledge representation language that allows the encoding of commonsense knowledge about the most diverse domains, and the definition of reasoning modules capable of planning, diagnostics, and learning.

Although A-Prolog can be used effectively to encode knowledge about complex domains, there is still a lack of well-established software engineering inspired tools and methodologies aimed at helping the programmer in this task. Most existing approaches [7, 6, 9, 4] involve a substantial language redesign, and need to tackle important issues involved in the design of modular extensions of non-monotonic formalisms. Finalizing the design of such a language, its implementation, and its spreading through the community, is still likely to require a considerable amount time.

In this paper, we propose a *light-weight* extension of A-Prolog, called *RSig*, introducing only a few simple constructs with straightforward semantics, and nonetheless providing key support for simple modular design of programs. It is our belief that *RSig* can be quickly learned and used by average A-Prolog users to both write new programs

and restructure existing programs, thus providing a first step towards the use of more sophisticated extensions of A-Prolog.

Drawing from our experience of encoding knowledge in A-Prolog, we have identified two main requirements that, we believe, need to be satisfied by any extension of A-Prolog aimed at simplifying the task of encoding complex knowledge bases:

1. It should be possible to develop portions of an A-Prolog program independently from each other.
2. In the inference engines that require typing of variables, such as LPARSE, the actions needed to provide such typing should interfere as little as possible with the programming task.

The first requirement involves the ability, frequently used in imperative programming, to define modules. Ideally, a module should be viewed by the module's users as a black-box, with clearly specified input and output. The module's users should be able to entirely disregard the actual implementation of the module.

If we turn our attention to the goal of limiting the burden of variable typing as much as possible, we see that, of the two most widely used inference engines, only DLV [5] satisfies this second requirement, because it does not *require* the typing of variables. However, if a programmer chooses to use variable typing for efficiency reasons, then he is forced to do that explicitly. Moreover, DLV still lacks the ability to work with function symbols, which substantially limits its applicability in the encoding of complex domains.

The requirement is not satisfied by LPARSE+SMODELS[1] [19, 16], as well as by the inference engines that rely on LPARSE (e.g. [13, 1, 14, 15]). In fact, with LPARSE, a programmer either explicitly types every variable, or uses the implicit typing facility provided by the $\#domain$ directive. Unfortunately, $\#domain$ fails to satisfy the requirement on typing: first of all, it forces the programmer to adhere to strict, and often unnatural, conventions on the use of variables; moreover, it forces the programmer to keep in mind one extra piece of information: the association between variables and their domains, with the consequence of interfering with the programming task; finally, it limits the ability of dividing a program in independent modules, because of the global scope of the $\#domain$ directive.

On the other hand, we believe that *RSig* satisfies both requirements above, and simplifies the task of representing knowledge for complex domains, by introducing only a small number of new constructs. The extension is based on the introduction of *signature declarations* and *module definitions*.

Although the main ideas behind *RSig* are substantially independent from a particular inference engine, here we concentrate on extending the language of LPARSE. The choice is motivated by the fact that LPARSE already allows function symbols, and that its sources are publicly available. An implementation of a parser for *RSig*, based on LPARSE, is available online from `http://krlab.cs.ttu.edu/~marcy/RSig/`.

[1] As here we are mostly concerned with language issues, rather than with inference algorithms, from now on we will refer to the pair LPARSE+SMODELS by the term LPARSE.

The paper is organized as follows. In the next section, we give an informal presentation of *RSig*. In Sections 3 and 4, we define the syntax and semantics of the language. In Section 5 we show an example of use of *RSig*. In the final sections, we discuss related work and draw conclusions.

2 *RSig*: The General Idea

Before we give a precise definition of *RSig*, let us describe the general idea behind the language.

As we mentioned above, *RSig* introduces signature declarations and module definitions. We call *signature declaration* of a function or relation the specification of the types of its arguments. The type of an argument is a sort – a unary predicate defined in the program. For example, let us specify the signature of a relation $sign(n, s)$ where n is an integer between given constants min and max, and s is -1, 0, or 1.

We begin by defining suitable sorts:

$$num(min..max).$$

$$sign\_type(-1).$$
$$sign\_type(0).$$
$$sign\_type(1).$$

The signature of $sign$ is given by a statement:

$$\#sig\ rel\ sign(num, sign\_type).$$

Its informal meaning is "relation $sign$ takes one argument of type num followed by one of type $sign\_type$." The keyword rel specifies that we are declaring the signature of a relation.

Avoiding explicit typing has substantial advantages in terms of program readability and writability, including the elimination of certain types of programming errors. As an example, let us see how relation $sign$ above can be defined with and without signature declarations.[2] Recall that, mathematically, the function "sign" can be defined as:

$$sign(n) = \begin{cases} 1 & \text{if } n > 0 \\ 0 & \text{if } n = 0 \\ -1 & \text{otherwise.} \end{cases}$$

The definition can be encoded in A-Prolog as:

$$sign(N, 1) \leftarrow N > 0.$$
$$sign(0, 0).$$
$$sign(N, -1) \leftarrow \text{not } sign(N, S), S \neq -1.$$

where the body of the last rule encodes "otherwise." Unfortunately, these rules cannot be used directly with LPARSE. In fact, the variables occurring in numerical expressions

[2] In this part of the paper, we do not consider the $\#domain$ directive of LPARSE. A discussion on $\#domain$ can be found in Section 6.

such as "$N > 0$" need to be explicitly typed. Variable S needs to be explicitly typed, too, because it occurs in the scope of default negation. The resulting LPARSE program is:

$$num(min..max).$$
$$sign\_type(-1).$$
$$sign\_type(0).$$
$$sign\_type(1).$$

$$sign(N, 1) \leftarrow num(N), N > 0.$$
$$sign(0, 0).$$
$$sign(N, -1) \leftarrow num(N), sign\_type(S), \text{not } sign(N, S), S \neq -1.$$

For rules that contain several variables, explicit typing substantially reduces the readability of the program, and increases the chances of errors due to mistakes in specifying the types.

Using the signature declarations of *RSig*, the definition of $sign$ becomes:

$$num(min..max).$$
$$sign\_type(-1).$$
$$sign\_type(0).$$
$$sign\_type(1).$$

$$\#sig\ rel\ sign(num, sign\_type).$$

$$sign(N, 1) \leftarrow N > 0.$$
$$sign(0, 0).$$
$$sign(N, -1) \leftarrow \text{not } sign(N, S), S \neq -1.$$

The resulting definition of $sign$ is arguably more natural and easier to read and the chances of mistakes in writing the program are smaller.

The information from signature declarations also affects the special atoms of LPARSE, i.e. those expressions of the form:

$$min\{p(X, Y) : q(X) : r(Y)\}max$$

and

$$min[p(X, Y) : q(X) : r(Y)]max$$

The typing information extracted from the signature declarations is used *for the condition part of the special atom*. Thus, the program:

$$q(0..3).$$

$$\#sig\ rel\ p(q).$$

$$\{p(X)\}.$$

is as an abbreviation of:

$$q(0..3).$$

$$\{p(X) : q(X)\}.$$

Let us now focus on module definitions. A module definition in *RSig* is a collection of *import/export declarations*, signature declarations, and statements from the language of LPARSE. Unless overridden by an import/export declaration, the interpretation of each relation and function in a module is independent from the interpretations used outside the module. For example, the program:

$$p \leftarrow \neg r.$$

$$\#module\ m1.$$
$$\neg r.$$
$$\#end\ module.$$

does not entail p, while of course the program consisting of $\{p \leftarrow \neg r.\ \ \neg r.\}$ does. This separation of interpretations allows to work on different parts of the program independently, as each module can be viewed as a black-box, of which only the import/export declarations need to be known. For example, relation r in module $m1$ above could be used as an auxiliary relation, whose meaning is independent from that of the relation r used in the first rule of the program.

The import and export declarations allow to make the interpretations of some relations and functions in a module coincide with those used outside the module. A relation or function occurring in the scope of an import or export declaration is called *global*. Intuitively, these statements specify respectively "input" and "output" relations and functions of the module. The distinction between import and export declarations has the purpose of improving the readability of the program: when a global relation or function is intended to occur in the head of a module's rules, it is be listed in an export declaration. Similarly, when it occurs in the body of a module's rules, it is to occur in an import declaration.

Thus, if the interpretations of the two occurrences of relation r in the program above are intended to coincide, we add an $\#export$ declaration to module $m1$. The program:

$$p \leftarrow \neg r.$$

$$\#module\ m1.$$
$$\#export\ rel\ r.$$
$$\neg r.$$
$$\#end\ module.$$

entails p. As with any module-based approach, the relations declared in the import and export statements should be carefully selected during the design phase, in order to avoid conflicts. We say that a relation r is *local* to a module m if literals formed by r occur in the rules of m, and r does not occur in an import/export declaration within m.

To help the debugging of programs, *RSig* also introduces a new variant of the $\#hide$ directive of LPARSE:

$$\#hide\ *.$$

The new directive can be used *only* inside modules. The intuitive meaning of such a statement occurring in a module m is that *all the literals formed by relations local to* m

are hidden in SMODELS*' output*, unless they are explicitly shown by a $\#show$ directive in m. For example, given the program:

$p \leftarrow$ not r.

$\#module\ m1.$
$r.$
$\#hide\ *.$
$\#end\ module.$

SMODELS displays:

Answer 1
Stable Model: p

Notice that whenever relations local to a module are displayed by SMODELS, they are prefixed by the name of the module. For example, given the program:

$p \leftarrow \neg r.$

$\#module\ m1.$
$\#import\ rel\ r.$
$\#export\ rel\ r.$
$\neg r.$
$q \leftarrow$ not $r.$
$t \leftarrow q.$
$\#hide\ *.$
$\#show\ q.$
$\#end\ module.$

SMODELS displays:

Answer 1
Stable Model: $p\ \neg r\ m1.q$

As the reader may have noticed, t, although true, is not displayed because of the $\#hide * $ directive in $m1$.

3 Syntax

Let us begin the definition of the syntax of *RSig* by summarizing the syntax of the language of LPARSE.[3]

In the language of LPARSE, terms, atoms, and literals are defined as in A-Prolog. A *special atom* is an expression of the form:

$$min\{l_1 : l_2 : l_3 : \ldots : l_k\}max$$

[3] For sake of simplicity, in this paper we consider a simplification of the language of LPARSE. However, our approach extends in a natural way to the full language.

or

$$min[l_1 : l_2 : l_3 : \ldots : l_k]max$$

where l_i's are literals and min, max are integers or variables.

An LPARSE rule, or *regular rule*, is an expression of the form:

$$l_0 \leftarrow e_1, \ldots, e_m, \text{not } l_1, \ldots, l_n.$$

where l_0 and e_i's are literals or special atoms, and l_i's are literals.

LPARSE directives, or *regular directives*, are expressions of the form:

$$\#show\ l_1, \ldots, l_n.$$
$$\#hide\ l_1, \ldots, l_n.$$

where l_i's are literals (the list may be empty).

A program in the language of LPARSE, or *regular program*, is a collection of *regular rules* and *regular directives*. Next, we describe the extensions of the language introduced by ***RSig***.

A *relation signature declaration* is a statement:

$$\#sig\ rel\ r_1(p_1^1, p_2^1, \ldots, p_{k_1}^1), \ldots, r_m(p_1^m, p_2^m, \ldots, p_{k_m}^m).$$

where r_i's are relations of arity k_i and p_j^i's are names of sorts. The informal meaning of the statement (for every i) is "the arguments of relation r_i are respectively of types p_1^i, p_2^i, ..., $p_{k_i}^i$." A *function signature declaration* is a statement:

$$\#sig\ func\ f_1(p_1^1, p_2^1, \ldots, p_{k_1}^1) \rightarrow p_0^1, \ldots, f_m(p_1^m, p_2^m, \ldots, p_{k_m}^m) \rightarrow p_0^m.$$

where f_i's are functions of arity k_i and p_j^i's are as above. The informal reading of the statement is "the arguments of function f_i are respectively of types p_1^i, p_2^i, ..., $p_{k_i}^i$, and terms formed by function f_i are of type p_0^i." The term *signature declaration* identifies both relation and function signature declarations.

A *relation import (resp., export) declaration* is a statement:

$$\#import\ rel\ r_1(\_, \_, \ldots, \_), \ldots, r_m(\_, \_, \ldots, \_).$$

or, respectively:

$$\#export\ rel\ r_1(\_, \_, \ldots, \_), \ldots, r_m(\_, \_, \ldots, \_).$$

where r_i's are relation symbols, and the number of anonymous variables "_" listed matches the arity of each r_i. The informal reading of the $\#import$ statement is "symbol r_1 denotes the same relation associated with symbol r_1 outside the module," and similarly for all r_i's and for the $\#export$ statement.

A *function import (resp., export) declaration* is a statement:

$$\#import\ func\ f_1(\_, \_, \ldots, \_), \ldots, f_m(\_, \_, \ldots, \_).$$

or, respectively:

$$\#export\ func\ f_1(\_, \_, \ldots, \_), \ldots, f_m(\_, \_, \ldots, \_).$$

where f_i's are function symbols. The informal meaning is similar to that of relation import and export declarations. By *import declaration* we mean both relation import and function import declaration. Similarly for *export declaration*.

A *module definition* (or *module* for short) is the sequence of statements:

$$
\begin{array}{l}
\#module\ \mu. \\
\iota_1 \\
\vdots \\
\iota_m \\
\rho_1 \\
\vdots \\
\rho_n \\
\#end\ module.
\end{array}
$$

where μ is a constant denoting the name of the module (the name of a module must be unique), ι_i's are optional import and export declarations, and ρ_i's are regular rules, regular directives (with the exception of directives $\#show.$ and $\#hide.$, which are not allowed in modules), signature declarations, or the new directive $\#hide\ *$. We denote the set $\rho_1, \ldots, \rho_n$ by $\Gamma(\mu)$. The relations listed in $\iota_1, \ldots, \iota_m$ are called *global relations of* μ, and are denoted by $\Theta(\mu)$. The literals from μ, formed by relations that are not in $\Theta(\mu)$, are called *local literals of* μ. The functions listed in $\iota_1, \ldots, \iota_m$ are called *global functions of* μ, and are denoted by $\Lambda(\mu)$. If global relations of μ occur in the head of the regular rules of $\Gamma(\mu)$, they must be listed in an export declaration. If they occur in the body of the regular rules of $\Gamma(\mu)$, they must be listed in an import declaration. Similarly for global functions. *For simplicity, from now on we assume that each predicate and function symbol is associated with a unique arity, and that the same symbol cannot denote both a predicate and a function.*[4]

An *RSig program* is a collection of regular rules, regular directives, signature declarations, and module definitions.

4 Semantics

We give the semantics of *RSig* programs by defining a mapping from *RSig* programs to programs in the language of LPARSE. We proceed in two steps: first we eliminate module definitions, and in the resulting program we introduce explicit typing for the arguments of the functions and relations for which signature declarations are given.

Intuitively, the elimination of module definitions is based on the addition of suitable prefixes to the occurrences of predicate and function symbols in a module.

Let μ be a module. The module-elimination of a function symbol f with respect to μ (denoted by f^μ) is f if f is a global function of μ, and $\mu.f$ otherwise. The module-elimination of a variable is the variable itself. The module-elimination of a term $t = f(t_1, \ldots, t_k)$, denoted by t^μ, is $f^\mu(t_1^\mu, \ldots, t_k^\mu)$.

[4] Our approach applies beyond these restrictions, thanks to the use of the "rel" and "func" keywords in signature and import/export declarations.

The module-elimination of a predicate symbol p with respect to μ (denoted by p^μ) is p if p is a global relation of μ, and $\mu.p$ otherwise. The module-elimination of an atom $p(x_1, \ldots, x_m)$ with respect to μ is: $p^\mu(x_1^\mu, \ldots, x_m^\mu)$. Similarly, the module-elimination of a literal $\neg p(x_1, \ldots, x_m)$ is $\neg p^\mu(x_1^\mu, \ldots, x_m^\mu)$. We denote the module-elimination of a literal l with respect to μ by l^μ.

The module-elimination of a special atom $min\{l_1 : l_2 : \ldots : l_k\}max$ is the special atom $min\{l_1^\mu : l_2^\mu : \ldots : l_k^\mu\}max$. The module-elimination of special atom c with respect to μ is denoted by c^μ.

The module-elimination of a regular rule, regular directive, or signature declaration ρ is obtained by replacing all literals, special atoms, and terms in ρ with their module-eliminations. The resulting statement is denoted by ρ^μ.

The module-elimination of a directive $\#hide\ *$ with respect to a module μ is a directive $\#hide\ l_1, l_2, \ldots, l_m$, where l_i's are all those local literals of μ, which do not appear in any $\#show$ directive of μ. For example, the module-elimination of $\#hide\ *$ in the program:

$p \leftarrow \neg r.$

$\#module\ m1.$
$\#import\ rel\ r.$
$\#export\ rel\ r.$
$\neg r.$
$q \leftarrow$ not $r.$
$t \leftarrow q.$
$\#hide\ *.$
$\#show\ q.$
$\#end\ module.$

is $\#hide\ t$.

The module-elimination of a module μ is the set

$$\Gamma'(\mu) = \{\rho^\mu \mid \rho \in \Gamma(\mu)\}.$$

The module-elimination of a program Π is obtained by replacing every definition of a module μ by $\Gamma'(\mu)$. The following proposition follows easily from the construction of the module-elimination of Π:

Proposition 1. *For every program Π, the module-elimination of Π contains no module definitions and no $\#hide\ *$ directives.*

The programs obtained by the module-elimination process are called *module-free programs*.

The next step of the translation consists in providing typing for the arguments of the functions and relations listed in the signature declarations.

Given a module-free program Π, $\Delta(\Pi)$ denotes the set of signature declarations from Π. For every predicate p or function symbol f such that, respectively, $p(s_1, s_2, \ldots, s_m)$ or $f(s_1, s_2, \ldots, s_k) \rightarrow s_0$ occur in $\Delta(\Pi)$, let δ_f^i denote s_i (recall that s_i's are names of unary predicates).

The *explicit-typing set* of a constant or variable is the empty set. The explicit-typing set of a term $t = f(t_1, \ldots, t_k)$ is denoted by t^σ, and consists of the set of atoms:

$$\{\delta_f^0(t), \delta_f^1(t_1), \delta_f^2(t_2), \ldots, \delta_f^k(t_k)\} \cup \bigcup_{1 \leq i \leq k} t_i^\sigma.$$

For example, given the declaration:

$$\#sig\ func\ g(r, s) \rightarrow u, h(q) \rightarrow r.$$

the explicit-typing set of term $g(X, Y)$ is $\{u(g(X, Y)), r(X), s(Y)\}$, and the explicit-typing set of $g(X, h(Z))$ is $\{u(g(X, h(Z))), r(X), s(h(Z)), r(h(Z)), q(Z)\}$.

The explicit-typing set of an atom $a = p(t_1, \ldots, t_k)$, denoted by a^σ, is the set:

$$\{\delta_a^1(t_1), \delta_a^2(t_2), \ldots, \delta_a^k(t_k)\} \cup \bigcup_{1 \leq i \leq k} t_i^\sigma.$$

The explicit-typing set of a literal $\neg a$ is a^σ. For example, given the declarations:

$$\#sig\ rel\ p(u, v).$$
$$\#sig\ func\ g(r, s) \rightarrow u, h(q) \rightarrow r.$$

the explicit-typing set of $p(X, Y)$ is $\{u(X), v(Y)\}$; the explicit-typing set of $p(g(X, Y), Z)$ is $\{u(g(X, Y)), v(Z), r(X), s(Y)\}$; the explicit-typing set of $p(g(X, Y), h(Z))$ is $\{u(g(X, Y)), v(h(Z)), r(X), s(Y), r(h(Z)), q(Z)\}$.

The explicit-typing set of a special atom $c = min\{l_1 : l_2 : \ldots : l_k\}max$ is $c^\sigma = l_1^\sigma$. For example, given $1\{p(g(X, Y), Z)\}2$ and the signature declarations from the previous example, the explicit-typing set is:

$$\{u(g(X, Y)), v(Z), r(X), s(Y)\}.$$

We can finally define the explicit-typing set of a regular rule. Given a regular rule ρ, let $lit(\rho)$ denote the set of literals from ρ (only the special atoms from ρ do not belong to $lit(\rho)$). The explicit-typing set of a regular rule ρ is the set

$$\rho^\sigma = \bigcup_{l \in lit(\rho)} l^\sigma.$$

For example, the explicit-typing set of the rule in the program:

$$\#sig\ rel\ p(u, v), w(r).$$
$$\#sig\ func\ g(r, s) \rightarrow u, h(q) \rightarrow r.$$

$$1\{p(g(X, Y), Z)\}2 \leftarrow w(h(Z)).$$

is:

$$\{r(h(Z)), q(Z)\}.$$

Intuitively, the explicit-typing set provides the typing information for the arguments of functions and relations. To complete the translation, we modify each rule by adding

to it the atoms from suitable explicit-typing sets. This operation is called explicit-typing, and is defined more precisely as follows.

The *explicit-typing* of a special atom $c = min\{l_1 : l_2 : \ldots : l_k\}max$ is the atom $c^\tau = min\{l_1 : l_2 : \ldots : l_k : p_1 : p_2 : \ldots : p_m\}max$, where $c^\sigma = \{p_1, p_2, \ldots, p_m\}$. For instance, the explicit-typing of special atom $1\{p(g(X,Y),Z)\}2$ from the example above is:

$$1\{p(g(X,Y),Z) : u(g(X,Y)) : v(Z) : r(X) : s(Y)\}2.$$

The explicit-typing of a regular rule ρ is the rule ρ^τ, obtained from ρ by replacing every special atom c with its explicit-typing c^τ, and by adding ρ^σ to the body of ρ^τ. For example, the explicit-typing of the rule in:

$$\#sig\ rel\ p(u,v), w(r).$$
$$\#sig\ func\ g(r,s) \rightarrow u, h(q) \rightarrow r.$$

$$1\{p(g(X,Y),Z)\}2 \leftarrow w(h(Z)).$$

is:

$$1\{p(g(X,Y),Z) : u(g(X,Y)) : v(Z) : r(X) : s(Y)\}2 \leftarrow w(h(Z)), r(h(Z)), q(Z).$$

Finally, the *explicit-typing of a module-free program Π* is the program Π^τ, consisting of:

- The explicit-typing of every rule from Π;
- All the regular directives of Π.

The following proposition follows directly from the above construction.

Proposition 2. *For every module-free program Π, the explicit-typing of Π is a regular program.*

The semantics of ***RSig*** associates every ***RSig*** program Π with the program obtained by applying module-elimination to Π, followed by explicit-typing. The resulting program is denoted by Π^λ. The following corollary holds:

Corollary 1. *For every* RSig *program Π, Π^λ is a regular program.*

5 Example of Use of *RSig*

To demonstrate the use of *RSig*, in this section we employ the new language to combine existing programs from the literature. Suppose we want to combine the *Military Example* from Section 4 of [12] with the *theory of intended actions* from [9]. Program Π_M from [12] consists of the declaration (refer to Section 6 for a discussion on $\#domain$):

$$\#domain\ step(T), agent(A), fluent(F), target(TAR), report\_id(R).$$

together with the set of rules R_M:

$$h(F,T) \leftarrow report(R,T), content(R,F,t), \text{not } problematic(R).$$
$$problematic\_agent(A) \leftarrow problematic(R), author(R,A).$$
$$h(destroyed(TAR), T+1) \leftarrow o(attack(TAR),T), \neg failed(attack(TAR),T).$$
$$\vdots$$

Axioms Π_I for intentions, on the other hand, include the declaration:

$$\#domain\ step(I), action(A).$$

together with the set of rules R_I:

$$occurs(A, I) \leftarrow intend(A, I), \text{not } \neg occurs(A, I).$$
$$intend(A, I1) \leftarrow next(I1, I), intend(A, I), \neg occurs(A, I), \text{not } \neg intend(A, I1).$$
$$\vdots$$

Combining Π_M and Π_I using only A-Prolog is non-trivial, because the programs are written rather differently. Key issues are: (1) variable A is used for both actions and agents; (2) relations o from Π_M and $occurs$ from Π_I must be connected; (3) Π_M and Π_I have to be inspected to ensure that the same predicate and function names are not used with different meanings. In general, the sets of rules being combined will need to be modified by hand, which is a time-consuming and error-prone task.

On the other hand, using *RSig*, the programs can be merged without changes to the existing rules. All that is needed is removing the $\#domain$ declarations, and adding suitable declarations of signatures and modules. The program combining Π_M and Π_I, outlined below, consists of: (1) signature declarations for relations and functions of global scope; (2) module $military$, containing R_M together with appropriate import/export declarations and signature declarations for local relations and functions; (3) module $intentions$, containing R_I together with import/export and signature declarations.

$$\#sig\ rel\ h(fluent, step), occurs(action, step).$$
$$\#sig\ rel\ failed(action, step).$$
$$\vdots$$

$$\#module\ military.$$
$$\#import\ rel\ occurs(\_, \_), failed(\_, \_).$$
$$\vdots$$
$$\#export\ rel\ problematic\_agent(\_).$$
$$\#export\ rel\ h(\_, \_).$$

$$\#sig\ rel\ o(action, step).$$

$$o(A, T) :- occurs(A, T).$$

$$R_M$$
$$\#end\ module.$$

$$\#module\ intentions.$$
$$\#import\ rel\ occurs(\_, \_), intend(\_, \_), next(\_, \_).$$
$$\#export\ rel\ occurs(\_, \_), intend(\_, \_).$$
$$\vdots$$
$$R_I$$
$$\#end\ module.$$

6 Related Work

The language of LPARSE includes a directive, $\#domain$, which aims at allowing implicit typing. Differently from the signature declarations presented here, $\#domain$ specifies an association between each *variable* and a type. Thus, a declaration:

$$\#domain\ r(X).$$

states that occurrences of X denote an object of type r. For simple cases, $\#domain$ is fairly effective. For example, it allows to write a definition of relation $sign$ that is as compact as the one in *RSig*:

$num(min..max).$
$sign\_type(-1).$
$sign\_type(0).$
$sign\_type(1).$

$\#domain\ num(N).$
$\#domain\ sign\_type(S).$

$sign(N, 1) \leftarrow N > 0.$
$sign(0, 0).$
$sign(N, -1) \leftarrow$ not $\ sign(N, S), S \neq -1.$

However, $\#domain$ directives apply to *all* the occurrences of a variable in the program. This substantially complicates the task of adding other rules, because the programmer needs to keep in mind the typing of all the variables already declared. Suppose, for example, that we were to use the above definition of $sign$ in a program that already contains a formalization of sets. Such a program could contain rules defining when a set is empty, similar to:

%% If O is a member of set S, then S has at least one member.
$at\_least\_one\_member(S) \leftarrow member(O, S).$

%% Set S is empty unless we know that S has at least one member.
$empty(S) \leftarrow$ not $\ at\_least\_one\_member(S).$

Unfortunately, the two sets of rules cannot be combined directly, because the $\#domain$ directive for variable S forces the domain of S to be $\{-1, 0, 1\}$ even in the rules about sets: the programmer needs to carefully rename the variables in either set of rules. If, instead, he is writing *new* rules, the programmer has to select carefully the variables, in order to match the intended argument types for the relations or functions he is using. Additional difficulties arise when special atoms are used in the program, as the occurrence, in these atoms, of variables from a $\#domain$ directive often yields unintended results. On the other hand, when writing rules in *RSig*, one only needs information about the argument types of relations and functions, different sets of rules can be more easily combined, and the signature declarations do not interfere with special atoms.

Various languages for the modular encoding of knowledge have been proposed in [7, 6, 9, 4]. All of these efforts are far more ambitious than *RSig*, in that they allows sophisticated definitions of classes or templates, including various degrees of the specification of object-oriented style inheritance and parametrization. We believe that learning and mastering these extensions requires a substantial effort. The goal of our work was to provide a simpler extension of A-Prolog that can be easily learned, mastered, and used for both new and existing programs.

7 Conclusions and Future Work

In this paper, we have presented an extension of A-Prolog satisfying the two main requirements for the simplification of the task of encoding complex knowledge bases.

We believe that the resulting language, *RSig*, is simple to learn for average A-Prolog users, and yet effective in satisfying those requirements.

An implementation of *RSig*, based on LPARSE, is available from `http://krlab.cs.ttu.edu/~marcy/RSig/`. With respect to the language described here, the implementation has the following limitations:

- The types used in signature declarations must be *domain predicates*.
- The parser does not check for duplicated module names.
- The parser does not check for directives $\#show.$ and $\#hide.$ occurring inside module definitions.
- Import and export declarations are allowed to occur anywhere inside a module definition.
- No error checking is done for improper import/export declarations, for example when a global relation is used in the head of a module's rules, but is not listed in an export directive.

In the future, we expect to assess the effectiveness and ease of use of *RSig* by encoding various complex knowledge bases. In this respect, we have already begun using *RSig* for a sophisticated intelligent system (partially covered in [2]) that applies deep reasoning to question answering in the context of natural language understanding.

8 Acknowledgments

The author would like to thank Michael Gelfond and Yana Maximova Todorova for their suggestions, and the anonymous reviewers for drawing attention to related works. This work was partially supported by NASA contract NASA-NNG05GP48G and by ATEE/DTO contract ASU-06-C-0143.

References

1. Marcello Balduccini. CR-MODELS: An Inference Engine for CR-Prolog. In *LPNMR 2007*, May 2007.

2. Marcello Balduccini and Chitta Baral. *Knowledge Representation and Question Answering*, chapter 21. Handbook of Knowledge Representation. Elsevier, 2006.
3. Chitta Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, Jan 2003.
4. Chitta Baral, Juraj Dzifcak, and Hiro Takahashi. Macros, Macro Calls and Use of Ensembles in Modular Answer Set Programming. In *Proceedings of ICLP-06*, pages 376–390, 2006.
5. Francesco Calimeri, Tina Dell'Armi, Thomas Eiter, Wolfgang Faber, Georg Gottlob, Giovanbattista Ianni, Giuseppe Ielpa, Christoph Koch, Nicola Leone, Simona Perri, Gerard Pfeifer, and Axel Polleres. The DLV System. In Sergio Flesca and Giovanbattista Ianni, editors, *Proceedings of the 8th European Conference on Artificial Intelligence (JELIA 2002)*, Sep 2002.
6. Francesco Calimeri, Giovanbattista Ianni, Giuseppe Ielpa, Adriana Pietramala, and Maria Carmela Santoro. A System with Template Answer Set Programs. In *JELIA 2004*, 2004.
7. Thomas Eiter, Georg Gottlob, and Helmuth Veith. Modular Logic Programming and Generalized Quantifiers. In *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97)*, volume 1265 of *Lecture Notes in Artificial Intelligence (LNCS)*, pages 290–309, 1997.
8. Michael Gelfond. Representing Knowledge in A-Prolog. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, volume 2408, pages 413–451. Springer Verlag, Berlin, 2002.
9. Michael Gelfond. Going places - notes on a modular development of knowledge about travel. In *AAAI Spring 2006 Symposium on Knowledge Repositories*, 2006.
10. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of ICLP-88*, pages 1070–1080, 1988.
11. Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, pages 365–385, 1991.
12. Nicholas Gianoutsos. Detecting Suspicious Input in Intelligent Systems using Answer Set Programming. Master's thesis, Texas Tech University, May 2005.
13. Yulia Lierler and Marco Maratea. Cmodels-2: SAT-based Answer Sets Solver Enhanced to Non-tight Programs. In *Proceedings of LPNMR-7*, Jan 2004.
14. Veena S. Mellarkod. Optimizing the Computation of Stable Models using Merged Rules. Master's thesis, Texas Tech University, May 2002.
15. Veena S. Mellarkod and Michael Gelfond. Enhancing ASP Systems for Planning with Temporal Constraints. In *LPNMR 2007*, pages 309–314, May 2007.
16. Ilkka Niemela, Patrik Simons, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, Jun 2002.
17. Monica Nogueira, Marcello Balduccini, Michael Gelfond, Richard Watson, and Matthew Barry. An A-Prolog decision support system for the Space Shuttle. In *PADL 2001*, pages 169–183, 2001.
18. Timo Soininen and Ilkka Niemela. Developing a declarative rule language for applications in product configuration. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, May 1999.
19. Tommi Syrjanen. Implementation of logical grounding for logic programs with stable model semantics. Technical Report 18, Digital Systems Laboratory, Helsinki University of Technology, 1998.