EMBEDDED DOMAIN-SPECIFIC LANGUAGES IN PROLOG

Tomaž KOSAR, Marjan MERNIK
Institute of Computer Science,
Faculty of Electrical Engineering and Computer Science, University of Maribor,
Smetanova 17, 2000 Maribor, Slovenia, tel. (+386 2) 220 7455, E-mail: {tomaz.kosar, marjan.mermik}@uni-mb.si

SUMMARY
A domain-specific language is a language tailored to a specific application domain and precisely capture the domain's semantics. It can be implemented by the traditional or by the embedded approach. While for embedding mainly functional languages are used, it is shown in the paper that Prolog is also suitable as a "host" language. The advantages of using Prolog in embedding are declarativeness, unification, nondeterminism, and "natural" looking syntax of domain-specific languages.

Keywords: *programming language design and prototyping, logic programming, embedding, Prolog*

1. INDRODUCTION

Programming languages are programmer's most basic tools. They can greatly increase programmer productivity by allowing them to write a high-scalable, generic, readable and maintainable code. In this regard, a domain-specific language, which is a programming language for solving problems in a particular domain and provides built-in abstractions and notations for that domain, is by no means an exception. Usually, domain-specific languages are small, more declarative than imperative, and more attractive than general-purpose languages for variety of applications because of:

- enhanced productivity, reliability, reusability, maintainability,
- easier verification,
- reduced semantic distance between the problem and the program.

Domain-specific languages can be implemented by the traditional approach where domain-specific syntax is designed and syntax-directed translator is written (from scratch or extended) or generated. An alternative to traditional approach to the implementation of domain-specific languages is by embedding. In embedding approach, a domain-specific language is implemented by extending an existing "host" language by defining specific abstract data types and operators. A problem in a domain then can be described with these new constructs. Hence, application engineer can become a programmer without learning too much of a "host" language. Therefore, the new language has all the power of a "host" language. Advantages of the embedding approach are:

- the development effort is not so high,
- it produce a powerful language since new features comes for free.

Although a "host" language can be any general-purpose language, a functional language is a very appropriate as a "host" language, as shown by many researchers [12, 13, 14]. This is due to functional language features such as expressiveness, lazy evaluation, high-order functions, strong typing with polymorphism and overloading. Therefore, many successful embedded domain-specific languages use a functional language as a "host" language [6, 21]. In much less extent as a "host" language an imperative or logic languages are used. It is shown in the paper that logic programming language Prolog is also a very suitable as a "host" language.

The organization of the paper is as follows. In section 2 introduction to domain-specific languages is given. The use of Prolog as a "host" language is described in section 3, followed by small example in section 4. Finally, related work and conclusion are described in section 5.

2. DOMAIN-SPECIFIC LANGUAGES

A domain-specific language is a language tailored to a specific application domain and precisely capture the domain's semantics. So far, domain-specific languages have been used in various domains such as graphics, financial products, description and analysis of abstract syntax trees, web computing, 3D animation, robot control, etc. These applications have clearly illustrated the advantages of domain-specific languages over general-purpose languages in areas such as productivity, reliability, maintainability and flexibility. However, the benefits of domain-specific languages are not for free. Without appropriate methodology and tools these costs can be higher than the savings obtained by using a domain-specific language for application development. Since, the cost of domain-specific language design, development and maintenance has to be taken into account, one of the main questions is "When and how to design and implement a domain-specific language?" [18]. When we want to improve productivity, reliability, reusability or enable end user programming in some narrow, but well-defined domain than a domain-specific language might be a solution and answer of the first part of this question. The development of a domain-specific language usually includes following phases: analysis, design, implementation and finally their use. In the analysis phase the problem domain is identified and domain knowledge has to be gathered. Then a domain-specific language is designed that concisely describe applications in the domain. The implementation phase can be done using one of the following approaches:

- the traditional approaches
  - the interpretation/compilation, where standard compiler tools can be used, or tools dedicated to the implementation of domain-specific languages,
  - the preprocessing or macro processing, where new constructs are translated to statements in the base language by a preprocessor,
  - the extensible compiler or interpreter, where a compiler or an interpreter is extended with new constructs; this is usually done by reflection mechanism.
- the embedding approach, where in an existing language user-defined operators are used to build a library of domain-specific operations.

Above steps show that the development of domain-specific languages is itself a significant software engineering task, requiring a considerable investment of time and resources. One might argue that the development of domain-specific languages should not differ much from the design and implementation of general-purpose languages, where any tool that generates a compiler or an interpreter from formal language specifications can be used for efficient and rapid development of domain-specific languages. As we can see such an approach is only one of the possible approaches. Other approaches (embedding, preprocessing, extensible compiler/interpreter) can be more efficient and attractive in particular cases. Other shortcomings of domain-specific languages, in addition to development costs, are:

- user training costs,
- tool support limitations (how to obtain a good integration of domain-specific languages with other software development tools is one of the open problems in domain-specific language research).

While advantages of embedding approach was already mentioned in the introduction, the advantages of traditional approach are:

- the syntax can be closed to notations used by domain experts,
- the good error reporting is possible,
- domain-specific optimizations and transformations are possible.

This approach has also following disadvantages:

- the development effort is high,
- the domain-specific language might be poorly designed,
- problems with language extensions,

which can be overcame when:

- the compiler/interpreter generator is used,
- the modular and extensible formal method for domain-specific language design is used.

3. PROLOG AS A "HOST" LANGUAGE

The effectiveness of Prolog as a language for rapid prototyping compilers and for developing scanner generators, parser generators and code generators has already been shown [2]. While in [2] only lexical and syntax part of language definition has been covered, the idea of using Prolog in implementing various formal semantic methods appear soon. Indeed, various formal methods for programming language descriptions such as attribute grammars, operational semantics and denotational semantics have been implementing using Prolog [1, 5, 8, 19, 24]. The advantages of Prolog basically stem from the use of unification and nondeterminism, and the price paid for the advantages are slower execution times. Various Prolog implementation of formal semantics method shows that Prolog is reliable tool for programming language development, design and prototyping [15, 17]. In the work [20] was shown that logic programming paradigm can additionally improve semantic expressiveness of attribute grammars. One of the benefits of formal methods is also the possibility of automatic compiler/interpreter generation. Attribute grammars are very suitable for this task and many compiler-compiler system exists. Some of them PANDA [7] and PROFIT [19] implements logical attribute grammars, which from attribute grammars automatically produced Prolog code.

The above approach can be seen as a traditional approach to implementation of domain-specific languages. As previously mentioned, Prolog is also a very suitable as a "host" language in the embedding approach. Its advantages over functional languages are:

- the syntax can be much closer to the notation used by domain experts,
- some domains are fully declarative and can not be easy realized with functional languages.

The syntax of a domain-specific language is a very important and should not be underestimated. The syntax should be as closed as possible to the notation used in a domain. In this regard Prolog has some advantages over Haskell. For example, in Haskell infix constructors must begin with a colon, while postfix functions can not be defined. In Prolog a prefix, infix and postfix user-defined operators can be defined, and almost all build-in operators can be redefined by the programmer, who can change also priority and associativity of operators. With operators we can make source program much more "natural" looking.

In the next section a simple domain-specific language is presented which is implemented in Prolog by embedding approach.

4. FDL - FEATURE DESCRIPTION LANGUAGE

Feature diagrams [3] are important part of the feature modeling and the domain analysis where commonalities, variabilities and dependencies between variable properties in the application domain are discovered. Domain-specific languages have to be designed in a manner to capture variable part of application domains (common features are fixed). Therefore, it is very important to find variable properties in an application domain. For this purpose feature diagrams are used in the domain analysis. In the work [23] a domain-specific language FDL (Feature Description Language) has been invented to describe feature diagrams. There are many benefits of the FDL such as: variability computation, constraint satisfaction, automatic mapping to UML diagrams and further automatic source code generation.

```
Car: all(carBody, Transmission, Engine,
        HorsePower, pullsTrailer?)
Transmission: one-of(automatic, manual)
Engine: more-of(electric, gasoline)
HorsePower: one-of(lowPower,
        mediumPower, highPower)
pullsTrailer requires highPower
includes pullsTrailer
```

Fig. 1 An example of the FDL program [23]

In the Figure 1 an example of the FDL program for a simple car is presented [23]. The program states that a car consists of a `carbody`, `Transmission`, `Engine`, and `HorsePower`. The last feature `pullsTrailer` is optional (indicated by character '?'). Features `Transmission`, `Engine`, and `HorsePower` are composite features consisting from sub-features, where exclusive (one-of) or non-exclusive (more-of) choice can be made. At the end of the FDL program also some constraints over features are given (`pullsTrailer requires highPower`). The meaning of the program presented in the Figure 1 is the following feature expression:

```
one-of(
   all(carBody, automatic, electric,
       highPower, pullsTrailer),
   all(carBody, automatic, electric,
       gasoline, highPower,
       pullsTrailer),
   all(carBody, automatic, gasoline,
       highPower, pullsTrailer),

   all(carBody, manual, electric,
       highPower, pullsTrailer),
   all(carBody, manual, electric,
       gasoline, highPower,
       pullsTrailer),
   all(carBody, manual, gasoline,
       highPower, pullsTrailer)
)
```

The FDL language [23] has been implemented using the traditional approach to the implementation of domain-specific languages. In this case, the meta-environment ASF+SDF [22] has been used to automatically produced the FDL interpreter from formal specifications. The grammar specification was written in about of 25 lines of a SDF code. The meaning of a FDL program (semantics) is given by the feature diagram algebra [23], which consists of normalization rules, variability rules (see Figure 2), expansion rules and satisfaction rules (part of these rules are presented in the Figure 3).

```
[V1] var(A)               = 1
[V2] var(F?)              = var(F) + 1
[V3] var(all(F, Ft))      = var(F) *
                            var(all(Ft))
[V4] var(all(F))          = var(F)
[V5] var(one-of(F, Ft))   = var(F) +
                          var(one-of(Ft))
[V6] var(one-of(F))       = var(F)
[V7] var(more-of(F, Ft)) = var(F) +
           (var(F)+1)* var(more-of(Ft))
[V8] var(more-of(F))      = var(F)
```

Fig. 2 Rules for computing variability of FDL specified in ASF [23]

```
[S3] is-element(A2, Fs) |
     is-element(A2, Fs')= false
=======================================
     sat(all(Fs, A1, Fs'),
         Cs A1 requires A2 Cs') = false
```

Fig. 3 Satisfaction rules for "requires" specified in ASF [23]

The FDL language is a nice example of a domain-specific language and can be very easily implemented using the embedded approach where logic language is used as a "host" language. In the Figure 4 the FDL language implementation by embedding in Prolog is presented.

Satisfaction rules are very easily expressed in Prolog; compare operator `requires` in the Figure 4 with the satisfaction rules in the Figure 3.

```
% FDL implementation in Prolog
:- op(400, xf, ?).
:- op(410, fx, includes).
:- op(410, fx, excludes).
:- op(410, xfx, requires).
:- op(410, xfx, excludes).

one-of([X|_], [X]).
one-of([_|Xs], X) :- one-of(Xs, X).

more-of(Xs, X):- one-of(Xs, X).
more-of([X|Xs], [Y|Ys]) :-
   one-of([X|Xs], [Y]),
   remove(Y, Xs, Zs), more-of(Zs, Ys).

remove(X, Xs, Xs) :-
   not(member(X,Xs)),!.
remove(X, [X|Xs],Xs)  :- !.
remove(X, [_|Xs], Ys) :-
   remove(X, Xs, Ys).

([X], [X])? .
( _ , [])? .

% user-defined operators
includes X/Xs       :- member(X, Xs).
includes [Y]/Xs     :- member(Y, Xs).
includes [Y|Ys]/Xs  :- member(Y, Xs),
   includes Ys/Xs.

excludes X/Xs :- not(includes X/Xs).

X/Xs requires Y/Ys :- member(X,Xs), !,
   member(Y,Ys).
_ requires _.

X/Xs excludes Y/Ys :- member(X,Xs), !,
   not(member(Y,Ys)).
_ excludes _.
```

Fig. 4 The FDL embedded in Prolog

The meaning of the following program (compare it to Figure 1) is:

```
% FDL program
carbody X      :- X = [carbody].
transmission X :- one-of([automatic,
                          manual], X).
horsePower X   :- one-of([lowPower,
                  medium, highPower], X).
engine X       :- more-of([electric,
                        gasoline], X).
pullsTrailer X :-
                  ([pullsTrailer], X)? .
car(X, Y, Z, W, V):-
             carbody X, transmission Y,
             horsePower Z,
             engine W, pullsTrailer V,
             includes pullsTrailer/V,
             pullsTrailer/V requires
             highPower/Z.

X = [carbody]
Y = [automatic]
Z = [highPower]
W = [electric]
V = [pullsTrailer] ;
X = [carbody]
Y = [automatic]
Z = [highPower]
W = [gasoline]
V = [pullsTrailer] ;

X = [carbody]
Y = [automatic]
Z = [highPower]
W = [electric,gasoline]
V = [pullsTrailer] ;

X = [carbody]
Y = [manual]
Z = [highPower]
W = [electric]
V = [pullsTrailer] ;

X = [carbody]
Y = [manual]
Z = [highPower]
W = [gasoline]
V = [pullsTrailer] ;

X = [carbody]
Y = [manual]
Z = [highPower]
W = [electric,gasoline]
V = [pullsTrailer]
```

Moreover, variability is very easily calculated in Prolog with the statement (compare it to Figure 2):

```
variability(N):-findall(X/Y/Z/W/V,
                car(X, Y, Z, W, V), L),
                length(L, N).
```

Checking under what conditions features satisfy constraints is also easily expressed in Prolog:

```
?- car([carbody],[manual],X,[gasoline],
       [pullsTrailer]).
X =[highPower]
```

As a future work authors [22] plan to extend the FDL language with more complex constraints where boolean expressions and relational operators will be added. In the embedded approach such extensions are for free and at no extra costs.

```
?- carbody X, transmission Y,
   horsePower Z, engine W,
   pullsTrailer V, (includes medium/Z;
   includes highPower/Z).
```

An extension of the FDL language would also be to express that many features require the same feature. In this manner the constraints can be much shorter. For example:

```
?- carbody X, transmission Y,
   horsePower Z, engine W,
   pullsTrailer V,
   [pullsTrailer/V, electric/W]
   requires highPower/Z.
```

To achieve this goal a simple change to the operator `requires` have to be done.

Despite that our Prolog implementation does not implement all of the functionality of the original FDL language it is shown that embedding in Prolog has some advantages when we want to extend the language by modest implementation effort. This is also important since the development of domain-specific language is usually just a part of some larger project with limited resources allocated to the development of domain-specific language.

5. RELATED WORK AND CONCLUSION

One of the research goals in programming languages is to develop concepts and tools to facilitate design and implementation of programming languages, general-purpose languages as for domain-specific languages. Such concepts and tools should not only simplify the construction of language-based tools [10, 11] such as compilers/interpreters, editors, debuggers, various analyzers etc., but should also aid in the design of high-quality languages [13]. Such high-quality language can be obtained if the language is designed with one of the formal methods such as denotational semantics, operational semantics or attribute grammars. Many researchers already advocated the use of formal methods in designing domain-specific languages [16, 18].

In the work [9] authors used the Horn logic denotational approach for specification, efficient implementation, and automatic verification of domain-specific languages. They used Prolog for denotational semantics implementation and Definite Clause Grammars (DCGs) for obtaining a parser of domain-specific language. Hence, both syntax and semantics of domain-specific language are expressed in the logical framework. More efficient implementation of domain-specific language can be further automatically derived by partial evaluation. An advantage of their approach is that verification of programs written in the domain-specific language can be automatically obtained. This work [9] is an interesting approach and can be classified as a traditional approach to the implementation of domain-specific languages.

An alternative approach to obtain a high-quality domain-specific language is by embedding. In [24] logical framework for embedding is described. Domain-specific language infrastructure (debugger, profiler, etc) can be automatically generated using logical framework. In the work [4] authors used logic facts to declare aspects (in a sense of aspect-oriented programming) and hence implements an aspect language by embedding in Prolog. Again the extension and modification of an aspect language were easy. Another benefit of this approach is that in such cases a programmer does not need to re-implement the aspect weaver.

The benefit of embedding approach is that programming features come automatically and for free. In the FDL example which is presented in the section 4 it was very easy to extend the language with new features. Our experience using Prolog as a "host" language was mostly positive. Its usability in this regard is comparable to Haskell, most often used functional language in embedding. The beneficial Prolog feature is also its ability to define new operators almost without restrictions. This partly reduce a disadvantage of embedded approach that the syntax is far from optimal. However, other disadvantages of embedded approach are still present when using Prolog as a "host" language:

- the bad error reporting,
- domain-specific optimizations and transformations are hard to achieve,
- efficiency.

REFERENCES

[1] B.R. Bryant and A. Pan. Rapid prototyping of programming language semantics using prolog. *In Proceedings of IEEE COMPSAC'89*, pages 439-446, 1989.

[2] J. Cohen and T. J. Hickey. Parsing and compiling using prolog. *ACM Transactions on Programming Languages and Systems*, 9(2):125-163, April 1987.

[3] K. Czarnecki and U. Eisenecker. *Generative Programming*. Addsion-Wesley, 2000.

[4] K. de Volder and T. D'Hondt. Aspect-oriented logic meta programming. In *Proceedings of Meta-Level Architectures and Reflection, Second International Conference, Reflection'99*, pages 250-272. Springer-Verlag, LNCS 1616, 1999.

[5] P. Deransart and M. Maluszynski. A grammatical view of logic programming. In *Proceedings of International Workshop on Programming Languages Implementation and Logic Programming, PLILP'88*, pages 219-251, 1988.

[6] C. Elliott. An embedded modeling language approach to interactive 3D and multimedia animation. *IEEE Transactions on Software Engineering, Special issue on domain-specific languages*, 25(3):291-308, May/June 1999.

[7] A. Feng, Y. Sugiyama, M. Fuji, and K. Torii. Generating practical prolog programs from attribute grammars. In *Proceedings of IEEE COMPSAC'87*, pages 605-612, 1987.

[8] G. Gupta. Horn logic denotations and their applications. In *The Logic Programming Paradigm: A 25 year perspective. Springer-Verlag*, 1999.

[9] G. Gupta and E. Pontelli. A horn logic denotational framework for specification, implementation, and verification of domain specific languages. Technical report, NMSU, 1999.

[10] J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. *ACM Sigplan Notices*, 35(3):39-48, March 2000.

[11] P.R. Henriques, M.J. Varanda Pereira, M. Mernik, M. Lenič, J. Gray, and H. Wu. Automatic generation of language-based tools using the LISA system. IEE Software, 152(2):54-69, 2005.

[12] P. Hudak. Modular domain specific languages and tools. In *Proceedings of the 5th International Conference on Software Reuse*, pages 134-142, 1998.

[13] S.N. Kamin. Research on domain-specific embedded languages and program generators. *Electronic Notes in Theoretical Computer Science*, 14, 1998.

[14] J. Kollár, J. Porubän, and P. Václavík. The Classification of Programming Environments, *Acta Universitatis Matthiae Belii*, 10:51-64, 2003.

[15] R. Laemmel and G. Riedewald. Prological language processing. In *Proceedings of the 1st Workshop on Language Descriptions, Tools and Applications, LDTA'01*. ENTCS Series, 44(2), 2001.

[16] S. Mauw, W.T. Wiersma, and T.A.C. Willemse. Language-driven system design. In *IEEE CD ROM Proceedings of 35th Hawaii International Conference on System Sciences*, 2002.

[17] M. Mernik, M. Črepinšek. Prolog and Automatic Language Implementation Systems. *Acta Electrotehnica et Informatica*. 5(3):42-49, 2005.

[18] M. Mernik, J. Heering, and A. Sloane. When and how to develop domain-specific languages. In *ACM Computing Surveys*, 37(4):To appear, December 2005.

[19] J. Paakki. Profit: A system integrating logic programming and attribute grammars. *Proceedings of International Workshop on Programming Languages Implementation and Logic Programming, PLILP'91*, pages 243-254, 1991.

[20] J. Paakki. Attribute grammar paradigms - a high-level methodology in language implementation. In *ACM Computing Surveys*, 27(2):196-255, 1995.

[21] J. Peterson, P. Hudak, and C. Elliot. Lambda in motion: controlling robots with Haskell. In *Proceedings of the 1st International Workshop on Practical Aspects of declarative Languages*, 1999.

[22] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Oliver, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-environment: A component-based language development environment. In *10th International Conference on Compiler Construction*, volume 2027, pages 365-370. Lecture Notes in Computer Science, Springer-Verlag, 2001.

[23] A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *CIT Journal of Computing and Information Technology, Special issue on Domain-Specific Languages, Eds.: R. Laemmel and M. Mernik*, 10(1):1-17, 2002.

[24] Q. Wang and G. Gupta. Rapidly Prototyping Implementation Infrastructure of Domain Specific Languages: A Semantics-based Approach. In *Proceedings of the ACM Symposium on Applied Computing, SAC'05*, pages 1419-1426, 2005.

BIOGRAPHIES

Tomaž Kosar received the BSc degree in computer science at the University of Maribor, Slovenia in 2002. He is currently a young researcher at University of Maribor, Faculty of Electrical Engineering and Computer Science. His research for PhD degree is concerned with design and implementation of domain-specific languages. His research interest in computer science include also domain-specific visual languages, compilers, refactoring, and unit testing. He is a student member of the IEEE.

Marjan Mernik received the M.Sc. and Ph.D. degrees in computer science from the University of Maribor in 1994 and 1998 respectively. He is currently an assistant professor at the University of Maribor, Faculty of Electrical Engineering and Computer Science. He is also an adjunct associate professor at the University of Alabama at Birmingham, Department of Computer and Information Sciences. His research interests include programming languages, compilers, grammar-based systems, grammatical inference, and evolutionary computations. He is a member of the IEEE, ACM and EAPLS.