Extending open compilers

David Krmpotić, Tomaž Kosar, Marjan Mernik, Viljem Žumer
University of Maribor, Faculty of Electrical Engineering and Computer Science
Smetanova ulica 17, 2000 Maribor, Slovenia
david.krmpotic@gmx.net, {tomaz.kosar, marjan.mernik, zumer}@uni-mb.si
tel: ++386-2-220-7458 fax: ++386-2-251-1178

March 7, 2005

Abstract

Domain-specific language implementation is in general not an easy task. This is the main reason that kept them from getting more attention and reaching the expectations of domain-specific language researchers. Implementation of a domain-specific language can be demanding, but also very rewarding. Once the hard work is done, we can profit a lot from the effort invested. The purpose of this paper is to present a domain specific language construction with extending the open compilers. Their source code – available, well organized and clear – gives the language designers a possibility to incorporate their domain specific constructs into the general-purpose language by extending its compiler. In this paper, we present extension of Mono C# compiler with representative domain-specific language.

1 Introduction

A common way of describing domain abstractions is by defining a library for a particular general-purpose language (GPL). However, GPLs usually do not contain appropriate syntactic structures to use in straightforward way. On the other hand, domain-specific languages (DSLs) are customly designed languages that target the specific domain. DSLs provide to the end-users (usually not very proficient in the GPLs), the appropriate abstractions and notations (textual or visual) to solve the problem from the observed domain. DSLs are therefore less general as GPLs, but at the same time more expressive in their domain. Implementation of a domain-specific language can be demanding, but also very rewarding. Once the hard work is done, we can profit a lot from effort invested. The good news is that "the hard work" actually is not that hard. The purpose of this paper is to present the DSL construction with extending an *open compiler*. Provided that the source code for the compiler is available and that its production rules are well organized and clear, the addition of the new domain specific constructs can be relatively easy.

In the DSL literature, GPL libraries are often mentioned as DSL competitors. On the contrary to DSLs, the GPL library is much easier and straightforward to implement, but less intuitive to use for the end-user. DSL requires more work, but solutions are much easier to use. In this paper, both approaches are presented and compared. However, the special focus is on the steps required to implement the DSL with an extensible compiler/interpreter implementation approach. We present extension of Mono C# compiler [7] with representative DSL - *Feature Description Language* [4].

The organization of the paper is as follows: Related work is discussed in Section 2. A DSL case study, is presented in Section 3. A construction of an GPL program using application library for the case study is presented in Section 4. Implementation of DSL using the extensible compiler approach is described in Section 5. Benefits of extending the compiler are described in Section 6. And finally, concluding remarks are summarized in Section 7.

2 Related work

Extending the existing programming language with a new DSL requires some means of accessing the compiler's internal structure. This is usually done by reflection [5]:

- introspection: examination of selected system internals,
- explicit invocation: enabling programs to explicitly invoke operations, bypassing syntax in some cases, and
- intercession: meta-level specialization that adds to or modifies the existing features.

However, only the last level throughly supports DSL implementation using extensible compiler approach. Recently, some aspect oriented languages [1, 8] have been implemented using extensible compiler/interpreter approach. Similar, but much more radical approach, is to add the required information to the compiler's source code. However, this approach brings some advantages, that are described later in the paper.

3 Case study

In order to show the extension of an open compiler, we need to choose the appropriate domain to solve. Feature Description Language (FDL) [4] was devised to serve as an example.

In FDL we can describe features and their hierarchical composition. Feature can be `atomic`, `optional`, `one-of`, `more-of` or `all`. Atomic feature is the basic construct, optional feature can be present or not, `one-of` means that exactly one of the child features can be present in the configuration. `More-of` is similar to `one-of`, except that arbitrary number of child features can be chosen from the set. `All` means that all the children have to be present.

Below you can observe a FDL program for a simple *Car composition* definition:

```
feature car = all("carBody", transmission, engine,
                  horsePower, opt("pullsTrailer"))
feature transmission = one_of("automatic", "manual")
feature engine = more_of("electric", "gasoline")
feature horsePower = one_of("lowPower", "mediumPower",
                            "highPower")
constraint "pullsTrailer" requires "highPower"
constraint include "pullsTrailer"
```

The meaning of the FDL program are all the possible configurations of a system obtained by four transformation steps: regularization, normalization, expansion and constraint satisfaction. Interested reader can find more details about transformation steps in [4]. Below you can find a solution of the *Car* program:

```
one-of(
  all(cB, automatic, electric,           highPower, pT)
  all(cB, automatic,           gasoline, highPower, pT)
  all(cB, automatic, electric, gasoline, highPower, pT)
  all(cB, manual,    electric,           highPower, pT)
  all(cB, manual,              gasoline, highPower, pT)
  all(cB, manual,    electric, gasoline, highPower, pT)
      )
              * cB = carBody
              * pT = pullsTrailer
```

Every *All* feature above represents a possible solution of our problem.

4 GPL approach

As we mentioned before, a common way to solve a problem is to define a library for a particular GPL with a corresponding API. This API represents an interface that acts as a mediator between DSL and the domain knowledge from the application library. To demonstrate the Car program in GPL, the application library for FDL was created. Below you can find a GPL program describing the Car program from previous section.

```
//CARBODY
AtomicFeature carBody  =
      FeureBuilder.BuildAtomic("carBody");

//TRANSMISSION
AtomicFeature automatic =
      FeatureBuilder.BuildAtomic("automatic");
AtomicFeature manual =
      FeatureBuilder.BuildAtomic("manual");
OneOfFeature transmission =
      FeatureBuilder.BuildOneOf();
transmission.AddFeature(automatic);
transmission.AddFeature(manual);

//ENGINE
AtomicFeature electric =
      FeatureBuilder.BuildAtomic("electric");
AtomicFeature gasoline =
      FeatureBuilder.BuildAtomic("gasoline");
MoreOfFeature engine =
      FeatureBuilder.BuildMoreOf();
engine.AddFeature(electric);
engine.AddFeature(gasoline);
```

```
//HORSEPOWER
AtomicFeature highPower =
      FeatureBuilder.BuildAtomic("highPower");
AtomicFeature mediumPower =
      FeatureBuilder.BuildAtomic("mediumPower");
AtomicFeature lowPower =
      FeatureBuilder.BuildAtomic("lowPower");
OneOfFeature horsePower = FeatureBuilder.BuildOneOf();
horsePower.AddFeature(highPower);
horsePower.AddFeature(mediumPower);
horsePower.AddFeature(lowPower);

//PULLSTRAILER
AtomicFeature pullsTrailer =
      FeatureBuilder.BuildAtomic("pullsTrailer");
OptFeature optPullsTrailer =
      FeatureBuilder.BuildOpt(pullsTrailer);

//CAR
AllFeature car = FeatureBuilder.BuildAll();
car.AddFeature(carBody);
car.AddFeature(new ReferenceFeature("engine"));
car.AddFeature(new ReferenceFeature("transmission"));
car.AddFeature(new ReferenceFeature("horsePower"));
car.AddFeature(optPullsTrailer);

//DEFINITNIONS
FeatureDefinitions def = new FeatureDefinitions();
def.Add("engine", engine);
def.Add("transmission", transmission);
def.Add("horsePower", horsePower);

//CONSTRAINTS
ConstraintContainer cons = new ConstraintContainer();
cons.Add(
     new RequiresConstraint(pullsTrailer, highPower));
cons.Add(new IncludeConstraint(pullsTrailer));

//obtain the meaning of a program
FDLManager fdl = new FDLManager(car);

//regularize (resolve references)
  fdl.Regularize(def);
//apply normalization rules
  fdl.Normalize();
//apply expansion rules
  fdl.Expand();
//apply constraints
  fdl.Satisfy(cons);
//print the results
  Console.WriteLine(fdl.GetResults());
```

As we can see, using the GPL for representing our domain specific problem is not optimal. Structure of the code differs a lot from the intuitive description of FDL presented in previous section.

5 Extensible compiler approach

Contrary to the GPLs, DSLs are more concerned with providing a simple syntax for representation of the domain knowledge. To develop a DSL one can choose from variety of implementation approaches as defined in [6]. One of the most demanding is *extensible interpreter/compiler approach*. Here, DSL developer requires access to the definition of the base language notation in order to *incorporate* the DSL syntax definition. Compilers, that allow incorporating new features to the base language, are called *open compilers* - Mono, C# compiler, is one of them.

Mono is an open source implementation of Microsoft .NET Framework that runs on Windows, Linux and various Unix platforms. .NET Framework is an environment for running programs containing special type of code called the Microsoft Intermediate Language (MSIL). That way greater portability is achieved while .NET programs can run everywhere the framework is available, independent of the underlying architecture. Other advantages that come from the managed code concept are garbage collection, type safety, array bounds and index checking, and so forth.

Under the Mono project, the open source compiler for C#, Mono C# compiler (MSC), was developed. MSC itself is a MSIL (interpreted) executable and thus able to run anywhere the framework (Mono or Microsoft's .NET) is present. MSC was written in C# and initially compiled with CSC (Microsoft's original C# compiler). Now MSC is able to compile itself.

Our goal was to incorporate FDL definitions somewhere inside the C# code. Below you can find general FDL notation incorporated in C#. Of course, the structure of a specification is differing substantially from the pure FDL code (see Section 3) – it needs a lot of opening and closing statements, structure that nicely integrates with the rules of C# notation.

```
begin_spec(<FDL_feature>, <result>)
  begin_features()
    <feature_definition>
    ...
  end_features()

  begin_constraints()
    <constraint_definition>
    ...
  end_constraints()
end_spec()
```

`FDL_feature` is the root feature, an entry point to our composition. By `<feature_definition>` we define features (atomic, all, one-of, more-of, optional). Constraints (`<constraint_definition>`) are defined in a separate optional section.

To achieve the incorporation, we add the FDL compiling capability to our Mono C# compiler. Mono 1.0.1 is chosen to integrate our language into C# notation. Mono parser is specified using Jay (Berkley Yacc parser generator [3]) modified to use C# for the semantic actions. The file `cs-parser.jay` contains the specification of the Mono compiler in standard LEX/YACC notation.

In Jay specifications, the lexical analyzer is invoked from file `cs-tokenizer.cs`, where hand-coded lexical analyzer is defined. All of the language constructs (keywords, identifiers, punctuation, etc.) are defined here. We add our keywords to the special hashtable as shown bellow:

```
AddKeyword("begin_spec", Token.BEGIN_SPEC);
AddKeyword("end_spec", Token.END_SPEC);
AddKeyword("feature", Token.FEATURE);
AddKeyword("all", Token.ALL);
...
```

Some C# predefined constructs, like IDENTIFIER (for feature names and atomic literals), are reused from the original C# notation. That demands some constraints to be put on the FDL notation. For example, if we wanted to say that feature names can only start with a capital letter, we would have to add the definition for UPPER_CASE_IDENTIFIER to the existing tokenizer. The philosophy is to use what is already defined in the base language.

FDL productions are added to the parser specifications. We have to define the production rules and the extension point – this is where our compiler gets aware of the new FDL construct definition. Before compiling the compiler, the mentioned .jay file has to be compiled by Jay ported to C#. This way the C# code for parsing is obtained from more readable .jay rules.

```
class_member_declaration
  : constant_declaration
  | method_declaration
  ...
  | FDL_declaration ;
```

We add the `FDL_declaration` non-terminal to `class_member_declaration` so that FDL can be defined inside every C# class.

As you can see from the partial FDL language productions below, Jay specifications use LEX/YACC convention, meaning that capital letters are used for terminal symbols while non-terminals use small caps.

```
FDL_declaration : BEGIN_SPEC OPEN_PARENS IDENTIFIER
opt_spec_variable CLOSE_PARENS
  opt_features_decl
  opt_constraints_decl
  END_SPEC OPEN_PARENS CLOSE_PARENS opt_semicolon
  ;
opt_features_decl :
  /* empty */
  | BEGIN_FEATURES
    OPEN_PARENS CLOSE_PARENS features_decl
    END_FEATURES
    OPEN_PARENS CLOSE_PARENS opt_semicolon
  ;
features_decl
  : feature_decl
  | features_decl feature_decl
  ;
feature_decl:
  FEATURE IDENTIFIER ASSIGN feature;
  ;
feature: ...
```

In Jay, semantic code is written in curly brackets before and after the rule. As the compiler itself is written in C#, we can reuse the already written FDL library with its corresponding API.

With added semantical rules, one of above-mentioned rules becomes:

```
feature_decl:
  FEATURE IDENTIFIER ASSIGN feature {
    Feature fe = (Feature)$4;
    FDLdefinitions.Add((string)$2, fe)
  };
```

"$n"**, where n is a natural number, denotes the n-th argument of the right hand production. When control goes to a production and eventually reaches a terminal, this terminal or some function of it is returned and propagated up to became our argument. The syntax used for returning the terminal is **"$$".

Further example:

```
feature: {
    feature_stack.Push(FeatureBuilder.BuildAll());
  }
  all_feature {
    $$ = feature_stack.Pop();
  }
  ...
  | atomic_feature {
      $$ = FeatureBuilder.
             BuildAtomic((string)atomic_feature);
  };
```

Here we can observe another detail of implementation of the semantical analysis. We use the simple stack to help us build the feature tree. Every time a composite feature is created, it is pushed on the stack, so the children

can be added to the current feature on the stack. This way we can go one or more levels down the hierarchy without forgetting exactly which feature is the parent of the current one.

After extending the compiler with FDL, we are able to write the following program. Class `FDLCar` is a C# class, but it contains domain-specific constructs to define a Car program.

```
class FDLCar {
begin_spec(car, strResults)
  begin_features()
    feature transmission= one_of ("automatic",
                                  "manual"     )
    feature engine      = more_of("electric",
                                  "gasoline"   )
    feature horsePower  = one_of("lowPower",
                                 "mediumPower",
                                 "highPower"   )
    feature car = all ("carBody", transmission,
                       engine,   horsePower,
                       opt("pullsTrailer")    )
  end_features()

  begin_constraints()
    constraint "pullsTrailer" requires "highPower"
    constraint include "pullsTrailer"
  end_constraints()
end_spec()

  static void Main() {
    Console.WriteLine (strResults);
  }
}
```

6 Benefits of extending the compiler

Big advantage of the extensible compiler/interpreter approach is that the resulting program is much easier to understand, modify and maintain for the end-user. DSL incorporation into GPL can increase programming efficiency and readability of the code.

Error reporting

Supporting domain-specific abstractions and notation is just a first step to build a DSL. Usually, this is the only feature the language developers provide to the end-users. This insufficiency can be bridged by providing programming tools, like inspectors, that are aware of domain-specific constructs. While other DSL implementation approaches, find this task very hard, extensible compiler/interpreter approach already contains needed infrastructure for building them.

For example, adding error reporting in open compilers, like in Mono, is very easy. DSL's program syntactical errors will be caught automatically by the scanning and parsing process while handling semantical errors has to be added manually. To support that, Mono compiler has defined the following static method:

```
Report.Error(string message, Location location)
```

which shows the information about the error and terminates the compiling process. For example, if the feature was referenced, but never defined, we can report that very easily. We save all the references to a given feature using the variable `lexer.location` available during parsing. If upon completing the process, the referenced feature remains undefined, we report the error with all the dangling references' locations.

Benefits of open compilers

Sometimes the open version of a given compiler is not available, but another open compiler for the same programming language exists. This is also the case with Microsoft and Mono C# compilers.

In case we want to use Microsoft C# compiler for the major part of our project, we can put our code with DSL in a separate DLL and compile it with Extended Mono C# compiler and then use it normally from our project. This way our module can be consumed from any C# compiler, because the generated intermediate language is the same and compilers are not aware which compiler was used to produce the code.

We can go a step further and actually use our DSL extended GPL from the Visual Studio.NET IDE. We put our DSL in comments so that the IDE's on-the-fly error detection does not complain. We also have to tell the IDE to run our program for removing the comments and then the MSC instead of CSC.

Advantages/disadvantages of extending the compiler

The DSLs are meant to be used by the end-users without a special knowledge of programming languages, but skilful in their specific domain. Looking from another prospective, DSLs can also provide GPL programmers with more natural notation when using specific application library. We found one good example in [2], where DSL for SWING graphical notation in Java is constructed.

In [2], MetaBorg tool helps to include a particular DSL notation back into a GPL by assimilating (not extending) it into the language i.e. domain-specific notation is translated into some existing APIs operations. As result of assimilation, GPL code is obtained.

When we extend the compiler once, we learn the steps and adding additional DSLs in no longer a problem. This brings idea of multiple-DSL incorporation into base language notation. This can be useful, but also very dangerous. Open compilers are vulnerable to the possibility of interfering with the base language. Furthermore, special care has to be taken when choosing the non-terminals for our DSL. Otherwise unwanted assimilation can occur between base language and DSL. Other problem is that our new keywords cannot longer be used as variable names in programs. We could isolate the modules with DSL code and compile them with extended compiler to a library.

When defining more than one DSL, same measures have to be taken. Keywords, non-terminal names, etc. of all the defined DSLs should not interfere. Solution would be to use the domain prefix to all the keywords for that DSL. For example "FDLFeature" instead of "feature".

Disadvantages mentioned above led to the idea of special "framework" construction. Its task is to make easy as possible to language developers to incorporate their DSLs to base language. This tool, can not be seen as a compiler generator, which main task is to write specifications for a DSL. In contrary, this tool should automate integration of and integrate a DSL to existing language compiler and replace existing with new, *DSL-aware* compiler.

7 Conclusion

Extending the compiler can prove beneficial if we want to simplify the process of writing the programs in the existing GPL.

If we need the DSL compiler as a stand-alone entity meant to be used by the specialists from the field, it could also be better to extend the compiler. By using the existing infrastructure, we can reuse the existing language constructs, like keywords, already defined in base language. Other main advantage of this is the ability to use GPL around DSL to complement it. In this case we reversed the logic. Now the domain experts can compile their DSL programs and get the results - not to the screen, but in some data structure. They can further process those results by using the GPL's functionality. There is no need for the developer of the DSL to provide that functionality.

By extending open-compilers and preserving the programming tools, like debuggers, with the domain specific information, we have also encountered possible ambiguities when extending the compiler with variety of DSLs. As a future work we are observing the possibilities and usefulness of constructing a tool for Mono C# compiler to support easier integration of domain-specific notation to the existing compiler.

References

[1] Kai Bollert. On weaving aspects. In *Proceedings of the Aspect-Oriented Programming Workshop*, 1999.

[2] Martin Bravenboer and Eelco Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04)*, pages 365–383, 2004.

[3] Robert Corbett. Berkeley yacc, 1990. http://dickey.his.com/byacc/byacc.html.

[4] A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002.

[5] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, 1991.

[6] M. Mernik, J. Heering, and A. Sloane. When and how to develop domain-specific languages. Technical report, University of Maribor, CWI Amsterdam, and Macquarie University, 2003. Draft.

[7] Mono. Online dictionary for computer and internet technology definitions, available at `http://www.mono-project.com/`, 2005.

[8] Lionel Seinturier. Jst: An object synchronization aspect for java. In *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP99*, 1999.