

Tool Support for Framework-Specific Language Extensions

Elizabeth Bjarnason and Görel Hedin

Dept of Computer Science, Lund University
Box 118, SE-221 00 Lund, Sweden
e-mail: {Elizabeth.Bjarnason | Gorel.Hedin}@dna.lth.se

Abstract. The conventions connected to the use of object-oriented frameworks can be described by framework-specific language extensions. The programmer is then aided in writing more correct programs. In an integrated structure-oriented language-design environment such language extensions can be supported internally.

1 Introduction

White box frameworks are known for being hard to use since they require detailed knowledge of the internal structure of the framework [Joh88], and that a number of programming conventions [Hed97] must be adhered to when using the framework. Failure to follow these conventions may lead to unpredictable errors which are often left undetected until run-time. Framework-specific language extensions which capture these conventions allow such errors to be detected and reported to the programmer before the program is executed. This is especially useful when working in an integrated programming environment since editing support for the framework-specific syntax and semantics can then be supplied. Also, in such an environment debugging can be supported in terms of the extended language rather than in terms of the internal code of the framework. The design and implementation of framework-specific extensions can be made easier by supplying support for such language extensions in an integrated structure-oriented language-design environment. The syntax, static-semantics, and code generation for the language extensions are then defined in terms of a base language. The proposed techniques for handling such language extensions are intended to be used in our language-design environment, APPLAB [Bja96, BHN97], to support the interactive design, development and use of framework-specific language extensions.

APPLAB currently supports the interactive development of languages. A language can be designed by editing a grammar description, and an example program can simultaneously be edited in the new (changing) language. The editor used, both for grammars and programs, is structure-oriented and based on grammar interpretation. That is, it interprets the current grammar descriptions in order to supply language-specific behaviour to the program editor. Editing is performed on the abstract syntax trees of the programs, and not at the text level. Text editing of subtrees is supported by invoking a grammar-interpreting parser. The static-semantics and code generation is expressed by standard AGs using an object-oriented specification language.

2 Support for Language Extensions

Figure 1 shows how the grammar for a base language, G_{BL} , is extended for a framework, FW. The framework is programmed in the base language, whereas the application program AP is programmed in the extended language G_{BL+FW} . The grammar for the extensions, G_{FW} , can access the framework to implement the code generation of the new language constructs. Because the extended language imports the base language, rather than copying its definition, changes made to the base language can be automatically incorporated into the extended language. Such changes of the base language may be fairly frequent in an interactive language-design environment such as APPLAB.

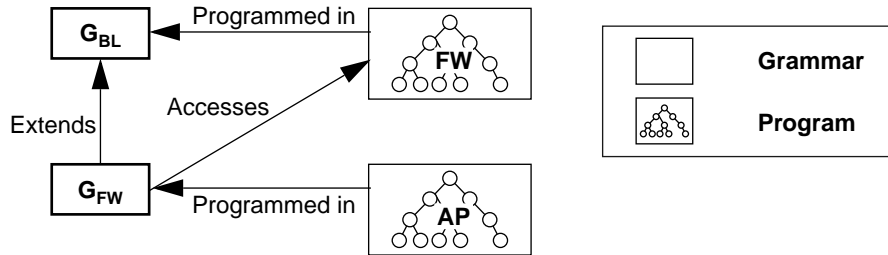


Figure 1. The dependencies between the language descriptions of the base language, G_{BL} and of the framework-specific language extensions, G_{FW} , the framework, FW, and an application program, AP, expressed in the extended language, G_{BL+FW} .

2.1 Subclassing Existing Language Constructs

When extending a language with new constructs it is desirable to reuse as much as possible of the existing implementation, as well as being able to add new features. In a declarative system using an object-oriented grammar notation this can be done by subclassing existing grammar rules. The new language construct then inherits the attributes and rules of the inherited grammar specification. New features can be added by defining additional attributes, and existing features can be modified by reimplementing the existing rules in the grammar specification of the new language construct.

Consider an example taken from robot programming where a base language is extended by adding a construct for moving the robot arm. The new construct, MoveTo, is declared as a subclass of the existing Statement declaration. Part of the specification of the extended language, G_{ROBOT} , is as follows:

```

MoveTo::=Statement ( "move" "to" Exp) (* Statement for moving the robot arm. *) (1)
{ (* expansion definition: (2)
  equations defining the expansion tree(see below) implementing the MoveTo
  construct in terms of the framework *)
(* static semantics: (3)
  equations which check that Exp is a Coordinate object (defined in the framework) *)
(* code generation: (4)
  equations which compute the code to generate by using the expansion tree*)
};
  
```

The abstract and concrete syntax (1) are specified, introducing the new keywords `move` and `to`, and stating that a `MoveTo`-statement contains an `Exp`-part. Static-semantic rules that ensure that the expression (`Exp`) represents a coordinate are added (3). The code generation for the `MoveTo`-construct (4) involves generating a call to the framework using the defined expansion tree (2).

2.2 Expansion Trees

A programmer using an extended language is only interested in seeing the new language constructs and their syntax. The system, on the other hand, needs to consider how the new constructs are implemented in terms of the base language and the framework, in order to correctly perform code generation and static-semantic checking. In a system whose internal representation of programs is based on abstract syntax trees, ASTs, *expansion trees* can be used for representing the new language constructs. Similarly to macros which are not expanded until compile time, expansion trees are not constructed until an attributed syntax tree is evaluated. This can be done by using *Higher-Order Attribute Grammars* [VSK89] which allow a node in the tree to be defined by the value of an attribute. We want such nodes to be invisible to the user, but used by the system to perform attribute evaluation, and thus code generation and static-semantic checking. Since the structure of an expansion tree follows the base language the system can evaluate its attributes in the same way as for the other parts of the program tree.

Part of the AST for a program using the extended language G_{ROBOT} is shown in Figure 2. The expansion tree connected to the `MoveTo`-node contains a procedure call of the `MoveLinear`-method of the framework. Note, that a reference back into the program AST is used to access the user-defined coordinates, while the other coordinates are defined by the language extension.

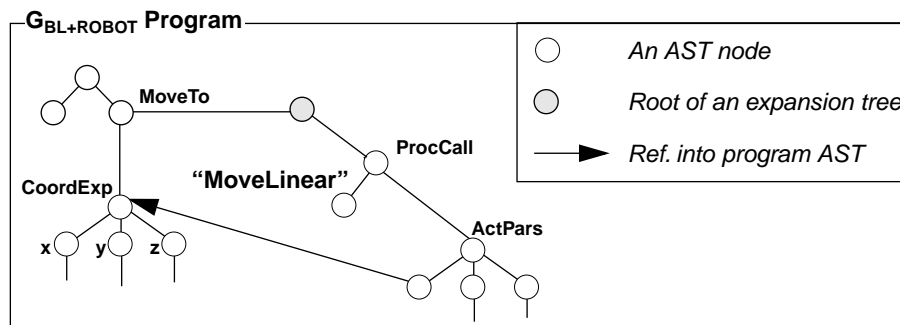


Figure 2. Part of the abstract syntax tree for a program expressed in the extended language $G_{\text{BL+ROBOT}}$.

3 Conclusions and Future Work

Framework-specific language extensions which supply framework-specific syntax and enforce the conventions of the framework make it safer and easier to use object-oriented frameworks. An object-oriented grammar notation allows new language constructs to be added by subclassing existing grammar rules. The new constructs can either reuse the properties, like syntax and semantics, of the existing language constructs, or specify new syntax, semantics etc.. When working in a programming environment which represents programs as abstract syntax trees, *expansion trees*, based on higher-order attribute grammars, can be used to implement new language constructs in terms of a framework. Static-semantic checking and code generation, as well as source code debugging, can then be supplied for the extended language. Such support for the design and implementation of framework-specific language extensions is being added to our language-design environment, APPLAB.

There are several interesting issues to look into connected to the use and implementation of framework-specific language extensions. For example, when a base language is changed this affects languages implemented as an extension of that base language, and programs expressed in the changed language. A mechanism is then needed for transforming the affected languages and programs into consistent versions according to the new version of the base language, by for example using techniques like those in the TransformGen system [GKL94]. It is also desirable to be able to allow multiple language extensions. That is, to combine several language extensions into one extended language. There may then be combinations of language constructs which contradict each other. Can such clashes be avoided or resolved automatically?

A lot of work remains to be done in this area. Both in implementing the proposed techniques and in doing further research into the area. Due to its declarative nature we believe APPLAB is a suitable platform for performing such research, and trying out new ideas in practice.

4 References

- [BHN97] E. Bjarnason, G. Hedin and K. Nilsson. APPLAB-An Application Language Laboratory. Techn Report, Dept. Computer Science, Lund University, 1997.
- [Bja96] E. Bjarnason. APPLAB: User's Guide (version 1.2). Techn. Report LU-CS-IR:96-01, Department of Computer Science, Lund University, 1996.
- [GKL94] D. Garlan, C. W. Krueger, and B. Staudt Lerner. TransformGen: Automating the Maintenance of Structure-Oriented Environments. *ACM TOPLAS*, 16(3):727-774, May 1994.
- [Hed97] G. Hedin. Attribute Extension - A Technique for Enforcing Programming Conventions. *Nordic Journal of Computing* 4(1997), 93-122. 1997.
- [Joh88] R. E. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22-35, June/July 1988.
- [VSK89] H. H. Vogt, S. D. Swierstra and M. F. Kuiper. Higher Order Attribute Grammars. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, ACM Sigplan Notices, 24(7), 1989.