Language Design and Implementation by Selection

Peter Pfahler, Uwe Kastens
Universität-GH Paderborn, Fachbereich 17, D-33095 Paderborn/Germany
Email: {peter, uwe}@uni-paderborn.de

Abstract

This paper demonstrates that design and implementation of languages for a specific domain can be done on a very high level of abstraction: elements and properties of a language are composed mainly by simple yes/no decisions. Consistency and completeness of the decisions are automatically checked. An implementation of the chosen language is obtained by selecting combinable specification components according to the user decisions and then feeding them into the language implementation system Eli. This approach is especially useful for designers of domain specific languages who usually have to deal with languages that have to be revised quite frequently and who normally do not have deep knowledge in the field of languages design and implementation.

1 Introduction

When studying a programming language or a language for a specific application domain one soon recognizes whether it is well designed, e.g. program objects and their properties are well chosen, they fit well to the operations defined for them, notation and structure reflect the semantics of the constructs, and it is easy to use with few exceptions and surprises. On the other hand it is very difficult to teach how to design a language such that it satisfies these requirements and suitably serves the intended purpose.

The implementation of the newly designed language adds another dimension to these difficulties, even if powerful state-of-the-art compiler construction tool sets are used. Such systems considerably raise the level in which language implementation specifications are formulated compared to manual or syntax-centered ("lex/yacc") approaches. Nevertheless, a considerable amount of knowledge and work is still required to implement a newly designed language.

This paper presents an approach to raise the degree of abstraction in designing and implementing domain specific languages to a very high level. Our approach is based on the following general reuse strategy:

> Restrict the problem domain. In our case: consider a specific class of languages, e.g. Pascal-like imperative languages or languages for data base report generators.
>
> Experts perform a domain analysis and produce a set of reusable and combinable components. In our case: language specification fragments and rules that control their composition.
>
> Users from the application domain can then solve their problems on a higher level of abstraction by selecting and combining precoined solutions. In our case: design and implement domain specific languages on a very high level of abstraction without language implementation experience.

Our language design system (internally called "LaLa" for "Language Laboratory", although this may not be its final name) consists of a user interface and a domain specific knowledge and specification base. The knowledge base contains a set of decisions and related questions and rules that ensure their completeness and consistency. The specification base consists of parametrized specification fragments for the Eli system [GHL+92, Eli96] being used as underlying language implementation machine. New domain specific language design systems are easily obtained by exchanging the knowledge and specification bases.

The language designer uses our system to interactively select language properties and elements. The design process mainly consists of simple yes/no decisions on the inclusion of certain language properties or components but it also allows for the free specification of some features such as error message texts. Examples for typical decisions are: Are variables explicitly declared or are they introduced implicitly? Which of the following predefined types shall your language provide? Do you want integer literals written in Pascal or C notation?

The system automatically adds implications, performs consistency and completeness checks and permanently informs the user about the state of the design process. An advisor component provides additional guidance, e.g. if you select the predefined type `int` it may be useful to have some operators (`add`, `mul`, ...) for integer values.

On request the system produces a set of language specifications for the compiler construction tool set Eli which generates an executable language processor for the newly designed language. So far we have implemented two design systems by instantiating our LaLa prototype with knowledge and specification bases for a tiny subset of Pascal ("MicroImperative") and for report generator languages on literature data bases ("RepGen"). For these two design systems the generated processors are a translator to C and an executable database report generator, respectively.

In the rest of the paper we present the structure of the language design system and give an idea of its use (section 2). Section 3 explains what kind of language design knowledge is captured in the knowledge and specification base for specific language domains. The development of our language design system heavily relies on important properties provided by the Eli system, especially the possibility to express language semantics independently from concrete notation. The use of Eli as a machine for language implementation is described in section 4. In the last section we relate our approach to other work in this area and present some ideas how it can be extended towards generating language descriptions, example programs, and to allow additional free-form descriptions.

2 The Language Design System

The design of the language design system was guided by the following objectives:

- The system should provide a clear view on the design space, i.e. on available language elements and on all possible alternative decisions concerning these elements.
- The system should support the language designer in his work by automatically computing consequences of user decisions and providing advice by suggestions for related decisions.
- The system should not restrict the designer's freedom to explore the available design space. The user should be able to investigate all subjects (even the ones have become irrelevant due to previous decisions) and should be allowed to redo any decision (even the ones that the system automatically added due to other decisions[1]).

[1] This should normally lead to an inconsistency, however.

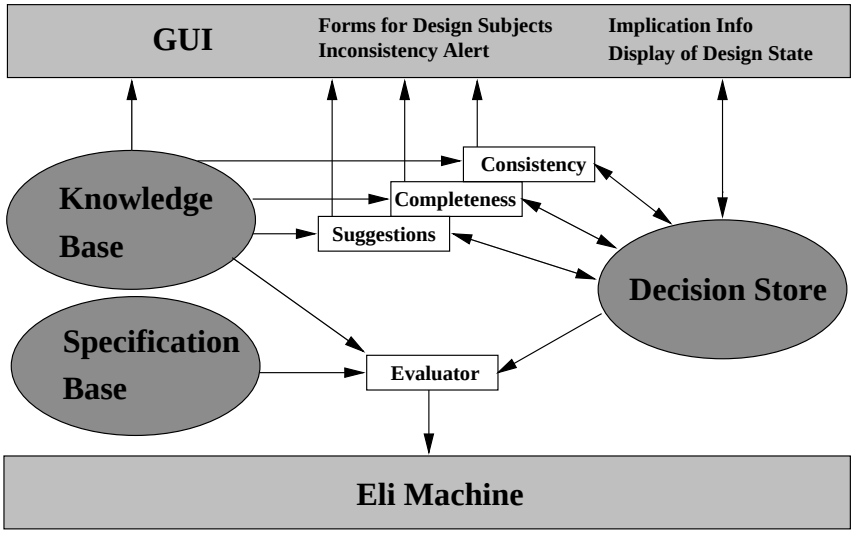

Figure 1: The Language Design System

- The system should tolerate temporary inconsistencies.
- The system should guarantee that only consistent and complete language specifications are passed to the underlying language implementation system. This point is especially important since the designer cannot be expected to deal with error reports from compiler generator systems referring to language specifications that he is not supposed to be aware of.
- The system should be domain independent, i.e. it should consist of a set of parametrizable GUI components and a fixed control kernel which are instantiated by providing domain specific knowledge and specification bases.

These design considerations have led to the system structure shown in Fig. 1. The knowledge base groups the design decisions into *subjects*, e.g. subject "statements" contains the decisions on the inclusion of "while", "ifthenelse" and "compound" statements. Each such subject corresponds to a button in the main system window[2]. Pressing a subject button displays information

[2]In our prototype implementation the knowledge base does not yet contain any GUI layout information. Therefore the main system window simply shows a row of such subject buttons.

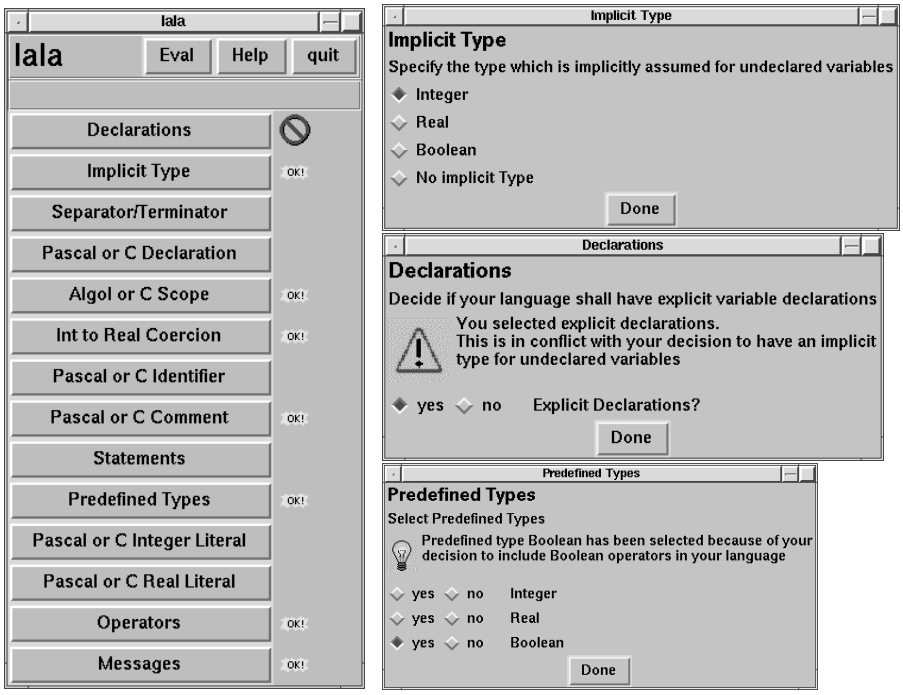

Figure 2: The MicroImperative System

about the subject and the state of decision binding. A dialog box is provided to make or change the associated decisions.

The state of all design decisions is recorded in the *decision store*. Once a decision value changes, e.g. the user decided to include 'while" statements, the implications of these decisions, e.g. "while" statement implies type "Boolean", are automatically added, if possible. The user is notified about the implicitly bound decisions. If the implication yields false, e.g. the user has explicitly excluded type "Boolean" before, an appropriate inconsistency message is displayed.

Similarly, consistency and completeness checks are performed and possible design suggestions are produced. The knowledge base provides the conditions for all these actions in the form of Boolean predicates over decision values (see section 3). Fig. 2 shows the main window for the MicroImperative example system and some of its subject windows.

Once the decision store is consistent and complete the designer may activate the evaluation step which applies the set of designer decisions to the specifi-

cation base. This step produces an input specification for the underlying Eli machine by transforming the set of user decision values into a set of switches and macro definitions which are used to instantiate the specification base components.

3 Knowledge and Specification Base Design

Using the language design system the designer of a domain specific language is freed from the task of specifying the language implementation. This makes this approach ideally suited for domain experts who are not language experts at the same time. The task of language implementation, however, is now moved to a new developer role, the *domain designer*. He is responsible for developing the system's knowledge and specification bases. His tasks include the investigation of the specific language domain, the analysis of variants and their combinability and the formulation of specification components for the Eli machine. The domain designer creates the knowledge and specification bases in the form required by the system kernel and tests the instantiated language design system, probably in cooperation with other domain experts.

The domain designer's task is a difficult one since most decisions are not isolated but dependent on other decisions. Legal combination of decisions have to be planned in advance and all potential conflicts have to be foreseen and formulated as predicates for the knowledge base. Here are some examples illustrating the domain designers work:

In the simplest case decisions are independent:

- Shall ';' be used as a separator or terminator for statements?
- What error message should be emitted if the data base cannot be opened?

In other cases different decision are related to each other:

- If implicit integer-to-real conversion is selected you should include both the predefined types `int` and `real` in your language (*implication*)!
- If you have not selected explicit variable declarations you don't need to specify the undefined-identifier error message text (*completeness*).

The domain specific knowledge base is developed by partitioning the design space into *subjects* and specifying for each subject

- its name
- an explanation text
- the decisions associated to this subject
- the decision type
  - TOGGLE: a single yes/no decision
  - ANYOF: a list of yes/no decisions
  - ONEOF: a list of alternatives (exclusive)
  - TEXTINPUT: a text entry facility, e.g. for specifying error message texts
- default values for the decisions
- the consistency rules: a list of conditions and implications including message texts for inconsistency alerts and informations which are emitted if implication rules implicitly influence other decisions
- the completeness rule: a Boolean predicate specifying when this subject can be regarded as completely handled.
- the suggestion rules: a list of Boolean predicates and associated texts
- the evaluation rule: specifies how decision values are translated into parameters for the specification base (see below).

All Boolean predicates are formulated over the set of all decision variables. The predicate that specifies completeness for the subject PascalDecl-CDecl ("do we have Pascal style or C style declarations") is e.g.

```
PascalDecl-CDecl.style != none || Declarations.HasDecl == no
```

meaning "The answer to this subject is satisfactory if the user selected Pascal or C style or the language does not contain explicit declarations".

The domain specific specification base is created by developing a set of specification components for the underlying Eli machine. The specification base is parametrized by switches and macro definitions which in most cases have a simple one-to-one correspondence to the decision variables in the knowledge base. This correspondence is specified by the evaluation rules. The system's evaluator module interprets these rules and produces an instantiated specification base as input for the Eli machine.

4 The Language Implementation Machine

The language design system uses the compiler construction environment Eli [Eli96] [GHL+92] as its language implementation machine. Eli combines a variety of standard tools that implement powerful compiler construction strategies into a domain-specific programming environment. Using this environment, one can automatically generate complete language implementations from application-oriented specifications. Eli offers complete solutions for commonly-encountered language implementation subtasks and contains libraries of reusable specifications [KW92], making possible the production of high-quality implementations from simple problem descriptions.

Apart from these general features of a powerful compiler construction environment there are some Eli characteristics which make it especially well suited for our purpose:

- Specifications can be decomposed in both dimensions by language elements as well as by the addressed implementation task. [KW91] demonstrates, for instance, how the name analysis subtask can be separated from other tasks and that it can be described independently from the syntactical structure of the language being implemented.
- Eli's input specification is a collection of typed specification components. There may, for instance, be several ".con" type specifications (describing concrete syntax fragments) instead of a single complete syntax specification.
- Eli allows to abstract the specification of semantic properties from concrete notation and from irrelevant parts of the abstract syntax. Semantic analysis and transformation tasks are performed on the abstract structure tree. The mapping between the concrete and abstract syntax is computed automatically by Eli such that modifications in the concrete notation which do not affect the abstract structure can easily be accomplished.

Due to these characteristics the Eli system can be directly used as the implementation machine of the design environment. The only necessary extension concerns the Eli input specifications in the specification base. They are parametrized by preprocessor switches and macros to select or deselect certain language features and insert e.g. user-contributed message texts.

These preprocessor directives are expanded by a conventional preprocessor before passing the specifications to Eli.

5 Related and Further Work

Domain specific languages are used by domain experts to describe their systems. Therefore such languages should be as close as possible to the user's conceptual view of the domain and easy to extend with new concepts. Furthermore, many domain experts postulate that they should own the application generation system to be independent from its manufacturer and to be able to adapt the system to changing requirements from the application domain. This, however, puts considerable requirements on the user-friendliness of the compiler techniques implementing the domain specific language [BH96].

Our approach goes one step further by letting the domain experts own the generator system that produces the application specific language implementation and providing him with a prefabricated set of combinable and parametrizable input specifications for this generator. User interface, knowledge base and the user advice system hide compiler technological aspects such that our system should be usable by application domain experts without computer science background.

The following fields of research relate to our work:

Language design [Wat90, Seb96] is primarily concerned with design issues for general purpose programming languages. General programming language design principles (like readability, writability, support for abstraction, reliability, simplicity and orthogonality) are certainly valuable for domain specific languages, too. For our approach language design considerations influence the specification base components as well as the knowledge base rules to combine these components.

Application Generators were introduced in [HKN85] for domain specific system descriptions in the domain of data base tools. [Cle88] generalizes the principle of application generators, points out their relation to compilers, and describes tools to construct them. In [Kas96] application generator construction using Eli is presented. Application generators are discussed in [Kru92] as one of the several general principles for software reuse.

Structure-oriented environments support the interactive development of domain specific applications by providing a graphical user interface integrating

a set of application specific tools (e.g. structure editor, graphical execution observation, incremental compilation). If such environments are generic, i.e. can be specialized to support different languages, and can be used on the Meta level, i.e. to specify the domain specific language the environment is to be used for, we speak of *Structure-Based Meta Environments*. Similar to our approach such systems can be used as language laboratories. The users can experiment with different design alternatives and can explore their consequences interactively in an evolutionary process. DOSE [FJS86] is an interactive prototyping environment for language design. It is based on a structure editor and uses formal specifications for syntax and semantics. ASF+SDF [Kli93] stands for "Algebraic specification formalism" and "Syntax definition formalism". Based on ASF+SDF a Meta-environment called GIPE has been built that allows for rapid prototyping of language specifications. The Mjølner Orm system has also been used as a structure-based Meta Environment [MM92]. The system APPLAB [Bja96] is a extension of Orm that especially aims at supporting interactive language development.

Structure-based Meta Environments have in common that they provide immediate feedback. This means that their users can freely switch between specifying the domain specific language and writing programs in that language using a structure editor. The key concept to this usage pattern is *immediate language interpretation* allowing for language dependent system behavior without intermittent generation and compilation phases. This concept is in contrast to our approach where the Eli system is used in a batch-oriented fashion[3] and where structure editors are not (yet) available. On the other hand, none of the systems mentioned above supports the "design by selection" aspect by providing domain specific specification and knowledge bases, which means that people using these systems for domain specific language design must be both domain and language experts.

For the near future we plan to extend our prototype language design system by the facility to automatically produce language reference manuals and example programs. These useful features should be rather easy to implement using a mechanism similar to the one that instantiates the specification base. We plan to further investigate the applicability of our approach by constructing further specification and knowledge bases for the subjects "Imperative Programming Languages" (domain: programming language education) and "Data Base Report Generators" (domain: data base tools).

[3] Due to its caching strategy Eli yields reasonable response time if specifications are locally modified.

Further, we plan to investigate how to provide support for domain specific language features which have not been foreseen in the knowledge and specification bases. Such language features require some kind of "free form specification" (as opposed to simple selection) which poses several challenging questions:

- How far can we go without forcing the system user to be a language implementation expert?
- Do we need additional high-level notations and tools for language specification[4] suitable for non-expert use?
- How can we still hide the underlying language implementation machine, i.e. can we still guarantee consistency to avoid low level error reports or, if not, how can we translate those reports back to the higher user level?

We believe that it is worthwhile to tackle these and related problems since the design and implementation of special purpose languages will certainly be a field of growing importance within software development.

References

[BH96] J. Bosch and G. Hedin. Editors's Introduction. In *Proceedings ALEL'96 Workshop on Compiler Techniques for Application Domain Languages and Extensible Language Models*, Technical Report LU-CS-TR:96-173. Lund University, Sweden, April 1996.

[Bja96] E. Bjarnason. APPLAB – A Laboratory for Application Languages. Technical Report LU-CS-TR:96-177, Lund Institute of Technology, Lund, Sweden, 1996.

[Cle88] J.C. Cleaveland. Building Application Generators. *IEEE Software*, pages 25–33, Jul 1988.

[Eli96] Eli Development Team. The Eli Home Page. http://www.uni-paderborn.de/fachbereich/AG/agkastens/eli_homeE.html, 1996.

[4] i.e. application generators in the domain of language specification

[FJS86] P. H. Feiler, F. Jalili, and J. H. Schlichter. An Interactive Prototyping Environment for Language Design. In *19th Annual Haiwaii International Conference on System Sciences, HICSS-19*, January 1986.

[GHL+92] R.W. Gray, V.P. Heuring, S.P. Levi, A.M. Sloane, and W.M. W aite. Eli: A Complete, Flexible Compiler Construction System. *Communications of the ACM*, 35(2):121–131, February 1992.

[HKN85] Ellis Horowitz, Alfons Kemper, and Balaji Narasimhan. A Survey of Application Generators. *IEEE Software*, pages 44–50, Jan 1985.

[Kas96] Uwe Kastens. Construction of Application Generators using Eli. In *Proceedings ALEL'96 Workshop on Compiler Techniques for Application Domain Languages and Extensible Language Models*, Technical Report LU-CS-TR:96-173. Lund University, Sweden, April 1996.

[Kli93] Paul Klint. A Meta-Environment for Generating Programming Environments. *ACM Transactions of Software Engineering and Methodology*, 2(2):176–201, March 1993.

[Kru92] Charles W. Krueger. Software Reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[KW91] Uwe Kastens and William M. Waite. An abstract data type for name analysis. *Acta Informatica*, 28:539–558, 1991.

[KW92] Uwe Kastens and William M. Waite. Modularity and Reusability in Attribute Grammars. Technical Report 102, Reihe Informatik, Universität-GH Paderborn, Fachbereich Mathematik-Informatik, July 1992.

[MM92] S. Minör and B. Magnusson. Using Mjølner Orm as a Structure-Based Meta Environment. Technical report, Department of Computer Science Lund University, 1992.

[Seb96] Robert W. Sebesta. *Concepts of Programming Languages*. Benjamin/Cummings, Redwood City, Calif., third edition, 1996.

[Wat90] David A. Watt. *Programming Language Concepts and Paradigms*. Prentice Hall, New York, 1990.