LISA: An Interactive Environment for Programming Language Development

Marjan Mernik, Mitja Lenič, Enis Avdičaušević, and Viljem Žumer

University of Maribor, Faculty of Electrical Engineering and Computer Science
Institute of Computer Science
Smetanova 17, 2000 Maribor, Slovenia

Abstract. The LISA system is an interactive environment for programming language development. From the formal language specifications of a particular programming language LISA produces a language specific environment that includes editors (a language-knowledgable editor and a structured editor), a compiler/interpreter and other graphic tools. The LISA is a set of related tools such as scanner generators, parser generators, compiler generators, graphic tools, editors and conversion tools, which are integrated by well-designed interfaces.

1 Introduction

We have developed a compiler/interpreter generator tool LISA ver 1.0 which automatically produces a compiler or an interpreter from the ordinary attribute grammar specifications [2] [8]. But in this version of the tool the incremental language development was not supported, so the language designer had to design new languages from scratch or by scavenging old specifications. Other deficiencies of ordinary attribute grammars become apparent in specifications for real programming languages. Such specifications are large, unstructured and are hard to understand, modify and maintain. The goal of the new version of the compiler/interpreter tool LISA was to dismiss deficiencies of ordinary attribute grammars. We overcome the drawbacks of ordinary attribute grammars with concepts from object-oriented programming, i.e. template and multiple inheritance [4]. With attribute grammar templates we are able to describe the semantic rules which are independent of grammar production rules. With multiple attribute grammar inheritance we are able to organize specifications in such way that specifications can be inherited and specialized from ancestor specifications. The proposed approach was successfully implemented in the compiler/interpreter generator LISA ver. 2.0 [5].

2 Architecture of the Tool LISA 2.0

LISA (Fig. 1) consists of several tools: editors, scanner generators, parser generators, compiler generators, graphic tools, and conversion tools such as fsa2rex, etc. The architecture of the system LISA is modular. Integration is achieved

R. N. Horspool (Ed.): CC 2002, LNCS 2304, pp. 1–4, 2002.

with strictly defined interfaces that describe the behavior and type of integration of the modules. Each module can register actions when it is loaded into the core environment. Actions are methods accessible from the environment. These actions can be executed via class reflection. Their existence is not verified until invocation, so actions are dynamically linked with module methods. The module can be integrated in the environment as a visual or core module. Visual modules are used for the graphical user interface and visual representation of data structures. Core modules are non-visual components, such as the LISA language compiler. This approach is based on class reflection and is similar to JavaBeans technology. With class reflection (java.lang.reflect.* package) we can dynamically obtain a set of public methods and public variables of a module, so we can dynamically link module methods with actions. When the action is executed, the proper method is located and invoked with the description of the action event. With this architecture it is also possible to upgrade our system with different types of scanners, parsers and evaluators, which are presented as modules. This was achieved with a strict definition of communication data structures. Moreover, modules for scanners, parsers and evaluators use templates for code generation, which can be easily changed and improved.

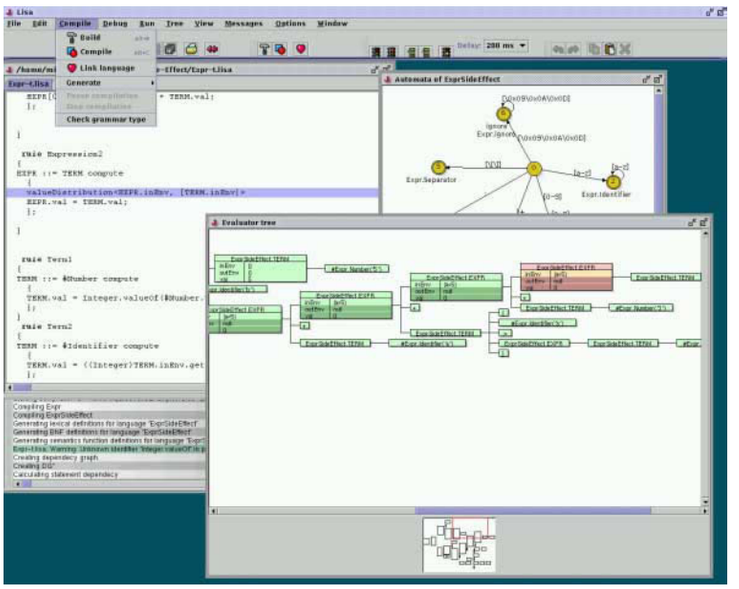

Fig. 1. LISA Integrated Development Environment

From formal language definition also editors are generated. The language-knowledgable editor is a compromise between text editors and structure editors since just colors the different parts of a program (comments, operators, reserved

words, etc.) to enhance understandability and readability of programs. Generated lexical, syntax and semantic analysers, also written in Java, can be compiled in an integrated environment without issuing a command to javac (Java compiler). Programs written in the newly defined language can be executed and evaluated. Users of the generated compiler/interpreter have the possibility to visually observe the work of lexical, syntax and semantic analyzers by watching the animation of finite state automata, parse and semantic tree. The animation shows the program in action and the graphical representation of finite state automata, the syntax and the semantic tree are automatically updated as the program executes. Animated visualizations help explain the inner workings of programs and are a useful tool for debugging. These features make the tool LISA very appropriate for the programming language development. LISA tool is freely available for educational institutions from: `http://marcel.uni-mb.si/lisa` . It is run on different platforms and require Java 2 SDK (Software Development Kits & Runtimes), version 1.2.2 or higher.

3 Applications of LISA

We have incrementally developed various small programming languages, such as PLM [3]. An application domain for which LISA is very suitable is a development of domain-specific languages. To our opinion, in the development of domain-specific languages the advantages of the formal definitions of general-purpose languages should be exploited, taking into consideration the special nature of domain-specific languages. An appropriate methodology that considers frequent changes of domain-specific languages is needed since the language development process should be supported by modularity and abstraction in a manner that allows incremental changes as easily as possible. If incremental language development [7] is not supported, then the language designer has to design languages from scratch or by scavenging old specifications. This approach was successfully used in the design and implementation of various domain-specific languages. In [6] a design and implementation of Simple Object Description Language SODL for automatic interface creation are presented. The application domain was network applications. Since the cross network method calls slow down performance of our applications the solution was Tier to Tier Object Transport (TTOT). However, with this approach the network application development time has been increased. To enhance our productivity a new domain-specific SODL language has been designed. In [1] a design and implementation of COOL and AspectCOOL languages has been described using the LISA system. Here the application domain was aspect-oriented programming (AOP). AOP is a programming technique for modularizing concerns that crosscut the basic functionality of programs. In AOP, aspect languages are used to describe properties, which crosscut basic functionality in a clean and a modular way. AspectCOOL is an extension of the class-based object-oriented language COOL (Classroom Object-Oriented Language), which has been designed and implemented simultaneously with AspectCOOL. Both languages were formally specified with mul-

tiple attribute grammar inheritance, which enables us to gradually extend the languages with new features and to reuse the previously defined specifications. Our experience with these non-trivial examples shows that multiple attribute grammars inheritance is useful in managing the complexity, reusability and extensibility of attribute grammars. Huge specifications become much shorter and are easier to read and maintain.

4 Conclusion

Many applications today are written in well-understood domains. One trend in programming is to provide software development tools designed specifically to handle such applications and thus to greatly simplify their development. These tools take a high-level description of the specific task and generate a complete application. One of such well established domain is compiler construction, because there is a long tradition of producing compilers, underlying theories are well understood and there exist many application generators, which automatically produce compilers or interpreters from programming language specifications. In the paper the compiler/interpreter generator LISA 2.0 is briefly presented.

References

1. Enis Avdičaušević, Mitja Lenič, Marjan Mernik, and Viljem Žumer. AspectCOOL: An experiment in design and implementation of aspect-oriented language. *Accepted for publications in ACM SIGPLAN Notices*. 3
2. Marjan Mernik, Nikolaj Korbar, and Viljem Žumer. LISA: A tool for automatic language implementation. *ACM SIGPLAN Notices*, 30(4):71–79, April 1995. 1
3. Marjan Mernik, Mitja Lenič, Enis Avdičaušević, and Viljem Žumer. A reusable object-oriented approach to formal specifications of programming languages. *L'Objet*, 4(3):273–306, 1998. 3
4. Marjan Mernik, Mitja Lenič, Enis Avdičaušević, and Viljem Žumer. Multiple Attribute Grammar Inheritance. *Informatica*, 24(3):319–328, September 2000. 1
5. Marjan Mernik, Mitja Lenič, Enis Avdičaušević, and Viljem Žumer. Compiler/interpreter generator system LISA. In *IEEE CD ROM Proceedings of 33rd Hawaii International Conference on System Sciences*, 2000. 1
6. Marjan Mernik, Uroš Novak, Enis Avdičaušević, Mitja Lenič, and Viljem Žumer. Design and implementation of simple object description language. In *Proceedings of 16th ACM Symposium on applied computing*, pages 203–210, 2001. 3
7. Marjan Mernik and Viljem Žumer. Incremental language design. *IEE Proceedings Software*, 145(2-3):85–91, 1998. 3
8. Viljem Žumer, Nikolaj Korbar, and Marjan Mernik. Automatic implementation of programming languages using object-oriented approach. *Journal of Systems Architecture*, 43(1-5):203–210, 1997. 1