Generating Interactive Programming Environments

J. Heering, P. Klint

1. Introduction

During the past ten years considerable progress has been made towards the automatic generation of interactive programming/development environments on the basis of a formal definition of some programming or specification language. In most cases, research has focused on the functionality and efficiency of the generated environments. These are the key quality factors which will ultimately determine the acceptance of environment generators. Only marginal attention has been paid to the actual development process of formal language definitions. Assuming the quality of automatically generated environments to become satisfactory within a few years, the cost of developing formal language definitions will then become the next limiting factor determining the ultimate success and acceptance of environment generators.

We will briefly sketch the design and implementation of a *meta-environment* (a development environment for formal language definitions) based on the formalism ASF+SDF and some of its applications.

2. Background—The Centaur system

A programming environment is a coherent set of interactive tools for developing and executing programs or specifications in some formal language. Well-known examples of such tools are syntax-directed editors, interpreters, debuggers, code generators, and prettyprinters. Programming environments

have been generated automatically for languages in such diverse application areas as programming, formal specification, proof construction, text formatting, process control, and statistical analysis. All projects in this area are based on the assumption that major parts of the generated environments are language independent and that all language dependent parts can be derived from a suitable language definition.

An example of a general architecture for programming environment generation is the CENTAUR system which was developed mainly by INRIA (France) in the ESPRIT GIPE project (1985-1993) in which CWI and the University of Amsterdam participated. This is a set of generic components for building environment generators. The kernel provides a number of useful data types but does not make many assumptions about, for instance, the language definition formalism itself. It has been extended with compilers for various language definition subformalisms as well as with several interactive tools. As such CENTAUR is an extensible toolkit rather than a closed system. We used it to build the ASF+SDF Meta-environment.

3. THE ASF+SDF META-ENVIRONMENT

The ASF+SDF Meta-environment [5] is a development environment for formal language definitions and an associated programming environment generator built on top of CENTAUR. Our research, which was part of the GIPE project mentioned before, went through three phases:

- Design of an integrated language definition formalism (ASF+SDF).
- Implementation of a generator for interactive programming/development environments given a language definition written in ASF+SDF.
- Design and implementation of an interactive development environment for the ASF+SDF formalism itself.

The result is the *Meta-environment* mentioned in section 1 in which language definitions can be edited, checked and compiled just like programs can be manipulated in a *generated environment*, which is an environment obtained by compiling a language definition. Note that 'compiling a language definition' and 'generating an environment' are synonymous in our terminology. Both the generator itself and the Meta-environment have been implemented on top of the CENTAUR system.

Figure 1 shows the overall organization of our system. First of all, we make a distinction between the *Meta-environment* and a *generated environment*. In the Meta-environment we distinguish:

- A language definition (in ASF+SDF) consisting of a set of modules `M1,...,Mn`.

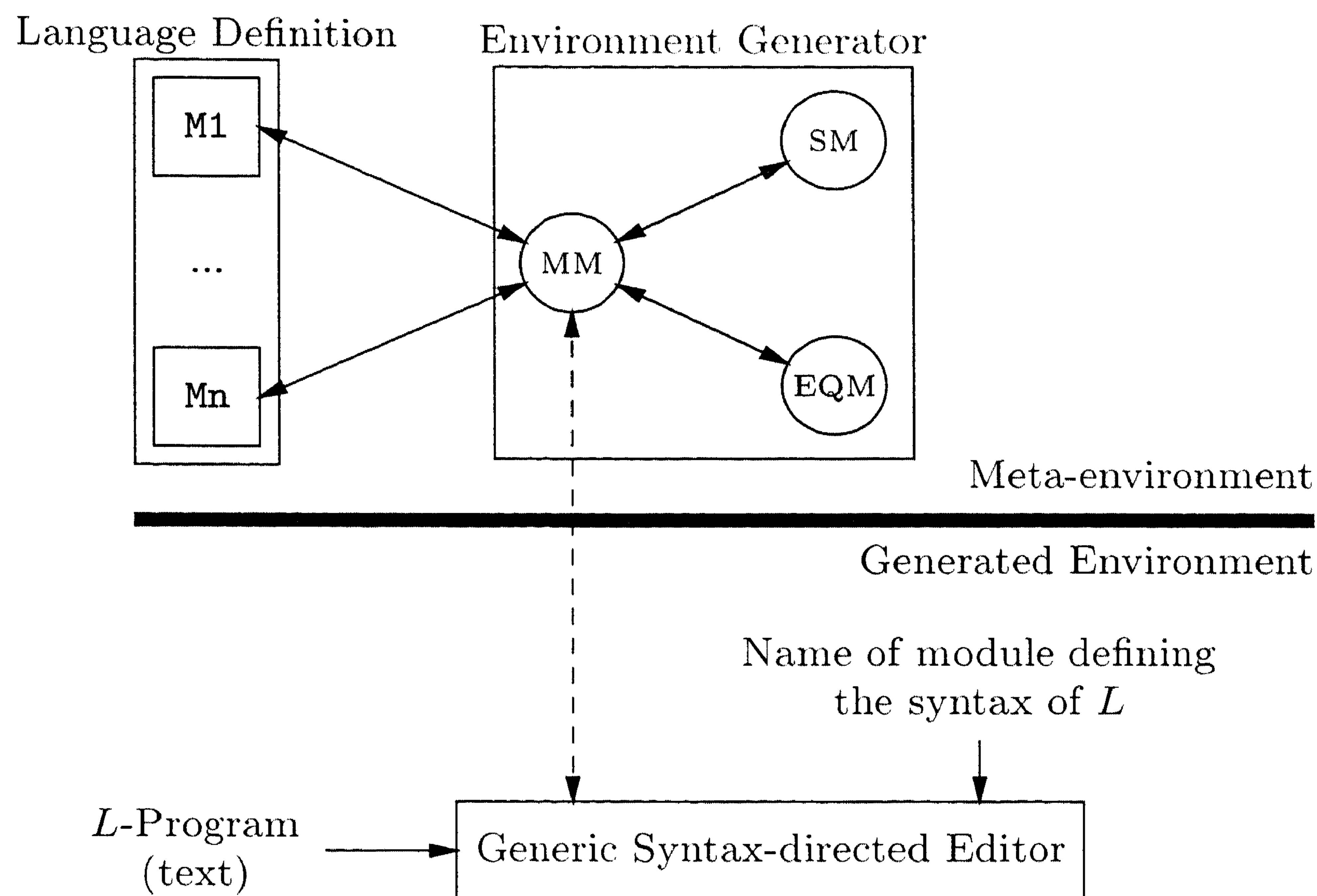

Figure 1. Global organization of the ASF+SDF Meta-environment.

- The environment generator itself, which consists of three components: a Module Manager (MM) controlling the overall processing of the modules in the language definition, the Syntax Manager (SM) controlling all syntactic aspects, and the Equation Manager (EQM), taking care of all semantic aspects of the language definition.

The output of the environment generator is used in conjunction with GSE (Generic Syntax-directed Editor), a generic building block which we use in generated environments. GSE not only supports text-oriented and syntax-oriented editing operations on programs but can also be extended by attaching 'external tools' which perform operations on the edited program such as checking and evaluation. The main inputs to the Generic Syntax-directed Editor are:

- A program text P.
- The modules defining the syntax of P.
- Connections with external tools.

As both the syntax description of P and the definition of external tools may be distributed over several modules, we are faced with the problem of managing several sets of syntax rules and equations simultaneously. One of the major contributions of the ASF+SDF Meta-environment is that the

system is so interactive and responsive that users are completely unaware of the fact that each modification they make to their language definition has major impacts on the generated environment. For instance, the presence of a parser generator is completely invisible to the user. As a consequence, the system is also accessible to 'naïve' users who have no previous experience with tools like scanner and parser generators. Important factors are: (1) an internal syntax tree representation ('term'—see next section) and a prettyprinter for the language are derived automatically from the language definition; (2) after parsing, syntax trees are built automatically; (3) the generated scanner, parser, tree constructor and rewrite system are interfaced automatically. To summarize, several parts of the generated implementation are derived from the language definition, and the system takes care of the interfacing of *all* components of the generated environment.

The implementation of the ASF+SDF Meta-environment is based on *lazy/ incremental* program generation [3].

4. TERM REWRITING

Central to our approach is the fact that we represent everything (i.e., programs and specifications being edited) as uniform tree structures which we call *terms*. All operations on programs—like checking and compiling—are expressed as operations on their underlying term representation. These operations have to be defined in the language definition and their execution is based on *term rewriting*. Given some initial term t_0, an attempt is made to apply a rule in the specification and transform the initial term into a new term t_1. This process is repeated until a term t_n is obtained to which no further rule is applicable. This is the *normal form* of the initial term t_0.

Clearly, efficient term rewriting is essential to us and we approach this problem from several angles. First, by investigating how rewrite rules can be translated directly to C programs. This would enable elimination of much of the overhead of term rewriting (in particular the search for matching rules) by performing an extensive static analysis of the given set of rules. A first prototype of this approach, the ASF2C compiler, has been completed and yields a speed improvement of a factor of 50–100 over our current, more interpretive, approach. Secondly, we have investigated *incremental rewriting*, a technique where previous runs of the rewriting engine on the same, or a slightly modified, term are being reused to avoid rewriting steps. This method is important for speeding up interactive tools that operate on terms. A typical example is an interactive typechecker operating on a program being edited by a user.

The fact that we base our computations on term rewriting gives us some interesting possibilities which can be exploited in the generated environments. One of them is *origin tracking* [7], which establishes links between subterms of the normal form t_n and the corresponding subterms (origins)

of the initial term t_0. This is vital information for interactive tools like error reporters (to associate an error message with a part of the source program) and animators (to visualize the statement we are currently executing). Generalizations of origin tracking (i.e., *dependence tracking*) permit the formulation of program slicing in the context of term rewriting. This may be useful in systems for interactive program understanding and reverse engineering.

5. CURRENT RESEARCH

The ASF2C compiler already mentioned above has demonstrated the potential of compiling algebraic specifications to efficient code. Its redesign, which is currently in progress, will introduce further optimizations and reduce the memory requirements of the generated code. Since the compiler has itself been specified in ASF+SDF, it also benefits from these improvements. Other extensions involve selective outermost rewriting [4], and the use of narrowing for simulating input/output.

In cooperation with J. Field (IBM T.J. Watson Research Center) work is in progress on optimizing compilers. The basic idea is to translate the source language, e.g., C, to an intermediate language called PIM. All further optimizations can be expressed as symbolic manipulations on the intermediate PIM representation of the program. These manipulations have been defined using ASF+SDF and are based on the ω-completion of algebraic specifications as described in [2].

In close cooperation with the University of Amsterdam (Programming Research Group) various extensions of the Meta-environment are being developed, e.g., generation of prettyprinters and documentation tools, visual editors, and the integration of parsing and rewriting. As a step towards reengineering the current implementation of the Meta-environment, a component interconnection architecture called TOOLBUS was developed in which all direct communication between components ('tools') is forbidden. Instead, all such communications are controlled by a process-oriented script that formalizes all the desired interactions between tools. No assumptions are being made about the implementation language or execution platform of each tool: tools may be implemented in different languages and may run on different computers. By adopting this approach we hope to make the implementation more flexible and manageable and to facilitate connecting externally developed software.

6. APPLICATIONS

Although originally designed as a generator for *programming* environments, it has turned out there are many other areas where the ASF+SDF Meta-environment can be applied. These range from general system design and the specification of environments for various languages to specific areas like

query optimization, hydraulic simulation, and application generators. We sketch three applications in some detail:

- In the context of ESPRIT project COMPARE (1991-1995), which aimed at the construction of optimizing compilers for parallel architectures, we designed a specification formalism fSDL for defining the intermediate data representations in compilers. In addition, using ASF+SDF we constructed a generator that compiled these specifications into C.
- In cooperation with a Dutch bank, we designed a specification language for financial products. Given such a product definition, appropriate (Cobol) code can be generated to include the information related to the product instance in the company's information system. In this way the time needed to construct software for new products can be reduced from months to days.
- In close cooperation with P.D. Mosses (Aarhus, Denmark), we constructed an interactive system to support the development of specifications written in *Action Semantics*, a formalism for defining the semantics of (programming) languages. It is currently being used for defining the semantics of ANDF (Architecture Neutral Definition Format), an exchange format for compiled programs.

Other applications using the ASF+SDF Meta-environment include:

- Automated induction proofs: D. Naidich (University of Iowa), T.B. Dinesh (CWI).
- Category theory: S. Vigna (University of Milano).
- Program transformations: M.G.J. van den Brand (UvA), H. Meijer (KUN).
- Message Sequence Charts: E.A. van der Meulen (UvA), S. Mouw (TUE).
- π-calculus: A. van Deursen (TUE).
- μCRL: J.A. Hillebrand (UvA), J.F. Groote (UU).

A survey of recent work can be found in [6]. Our earlier work on algebraic specifications can be found in [1].

ACKNOWLEDGEMENTS
The following persons made contributions to this project: H.C.M. Bakker, J.A. Bergstra, M.G.J. van den Brand, A. van Deursen, N.W.P. van Diepen, T.B. Dinesh, C. Dik, H. van Dijk, J.J. Ganzevoort, P.R.H. Hendriks, J.F.Th. Kamperman, A.S. Klusener, J.W.C. Koorn, M.H. Logger, E.A. van der Meulen, P.A. Olivier, J.G. Rekers, M. Res, F. Tip, S. Üsküdarli, A. Verhoog, E. Visser, S. van Vlijmen, P. Vriend, H.R. Walters, A. van Waveren.

REFERENCES

1. J.A. BERGSTRA, J. HEERING, P. KLINT (1989). *Algebraic Specification*, ACM Press Frontier Series. The ACM Press in cooperation with Addison-Wesley.
2. J. HEERING (1986). Partial evaluation and ω-completeness of algebraic specifications. *Theoretical Computer Science 43*, 149-167.
3. J. HEERING, P. KLINT, J. REKERS (1994). Lazy and incremental program generation. *ACM Transactions on Programming Languages and Systems 16(3)*, 1010–1023.
4. J.F.TH. KAMPERMAN, H.R. WALTERS (1995). Lazy Rewriting and Eager Machinery. JIEH HSIANG (ed.). *Rewriting Techniques and Applications, 6th International Conference (RTA-95), Lecture Notes in Computer Science*, Vol. 914, Springer-Verlag, 147-162.
5. P. KLINT (1993). A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology 2(2)*, 176-201.
6. M.G.J. VAN DEN BRAND, A. VAN DEURSEN, T.B. DINESH, J.F.TH. KAMPERMAN, E. VISSER (eds.) (1994). *ASF+SDF'95*, a workshop on Generating Tools from Algebraic Specifications. Tecnical Report P9504, Programming Research Group, University of Amsterdam.
7. A. VAN DEURSEN, P. KLINT, F. TIP (1993). Origin tracking. *Journal of Symbolic Computation 15*, 523–545.