UNIVERSITY OF AMSTERDAM

UvA-DARE (Digital Academic Repository)

A meta-environment for generating programming environments

Klint, P.

Published in:
ACM Transactions on Software Engineering and Methodology

Link to publication

Citation for published version (APA):
Klint, P. (1993). A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, *2*(2), 176-201.

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.

UvA-DARE is a service provided by the library of the University of Amsterdam (http://dare.uva.nl)

Download date: 27 Feb 2021

A Meta-Environment for Generating Programming Environments

PAUL KLINT
CWI, Amsterdam and the University of Amsterdam

Over the last decade, considerable progress has been made in solving the problems of automatic generation of programming/development environments, given a formal definition of some programming or specification language. In most cases, research has focused on the functionality and efficiency of the generated environments, and, of course, these aspects will ultimately determine the acceptance of environment generators. However, only marginal attention has been paid to the development process of formal language definitions itself. Assuming that the quality of automatically generated environments will be satisfactory within a few years, the development costs of formal language definitions will then become the next limiting factor determining ultimate success and acceptance of environment generators.

In this paper we describe the design and implementation of a meta-environment (a development environment for formal language definitions) based on the formalism ASF + SDF. This meta-environment is currently being implemented as part of the Centaur system and is, at least partly, obtained by applying environment generation techniques to the language definition formalism itself. A central problem is providing fully interactive editing of modular language definitions such that modifications made to the language definition during editing can be translated immediately to modifications in the programming environment generated from the original language definition. Therefore, some of the issues addressed are the treatment of formalisms with user-definable syntax and incremental program generation techniques.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*languages*; D.2.6 [**Software Engineering**]: Programming Environments; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*syntax, semantics*; D.3.4 [**Programming Languages**]: Processors

General Terms: Design, Languages

Additional Key Words and Phrases: Algebraic specification, application generators, application languages, concrete and abstract syntax, incremental program generation, language definition formalism, meta-environment, programming environment generation, programming language semantics, user-definable syntax

Partial support received from the European Communities under ESPRIT project 2177 (Generation of Interactive Programming Environments II–GIPE II). This is a completely revised and extended version of a paper that appeared earlier in *Algebraic Methods II. Theory, Tools and Applications*, vol. 490. J. A. Bergstra and L. M. G. Feijs, Eds., Lecture Notes in Computer Science, Springer-Verlag, 1991, 105–124.

Author's address: CWI (Centrum Voor Wiskunde en Informatica), P. O. Box 4079, 1009 AB Amsterdam, The Netherlands. email: klint@cwi.nl

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1993 ACM 1049-331x/93/0400-0176 $01.50

1. INTRODUCTION

Over the last decade, several research projects have focused on the automatic generation of programming environments given a formal specification of a desired language (for instance, Mentor [6], PSG [1], Synthesizer Generator [27], Gandalf [10], GIPE [14], Genesis [8], Graspin [7], and Pan [2]). A programming environment is here understood as a coherent set of interactive tools such as syntax-directed editors, debuggers, interpreters, code generators, and pretty printers to be used during the construction of texts in the desired language. This approach has been used to generate environments for languages in different application areas such as programming, formal specification, proof construction, text formatting, process control, and statistical analysis. All these projects are based on the assumption that major parts of the generated environment are language independent and that all language-dependent parts can be derived from a suitable high-level formal specification. Various problems have been studied:

—integration of text-oriented editing and syntax-oriented editing;
—automatic generation of incremental tools from nonincremental specifications;
—a single integrated language definition formalism versus several separate formalisms;
—generation of interpreters and compilers;
—fixed versus user-definable user interfaces;
—fixed versus user-definable logic in language definition formalisms;
—descriptive power of the language definition formalism (specification of polymorphic type systems, concurrency, etc.).

As a general observation, systems with fixed, built-in solutions for some of the problems mentioned above are very easy to use in the application area they were designed for—and probably in some unanticipated areas as well—but it may be difficult or even impossible to use them in other areas. Therefore, we currently see a trend toward systems with more open architectures consisting of cooperating sets of replaceable components. In this way one can obtain very general and flexible systems.

An example of such a general architecture is the Centaur system [4] developed in the GIPE project. It can be characterized as a set of generic components for building environment generators. These generic components support, among other things, operations for

—manipulating abstract syntax trees, and
—creating graphical objects and user interfaces.

The kernel thus provides a number of useful data types but does not make many assumptions about, for instance, the logic underlying the language definition formalism. This generality is achieved by permitting a simple interface between the kernel and logical engines such as a Prolog interpreter

or a rewrite rule interpreter. Note that these logical engines are not generated from specifications but are implemented separately.

The kernel has already been extended with compilers for various language definition subformalisms such as METAL [21], SDF [13], and TYPOL [5, 20], as well as interactive tools such as the structure-oriented editor CTEDIT, the generic syntax-directed editor GSE with integrated text-oriented and syntax-oriented editing capabilities, and a tool for controlling the execution of TYPOL specifications. The system thus resembles an extendible toolkit rather than a closed system.

The current Centaur system gives some support for the interactive development of language definitions (e.g., the interactive editing and debugging of TYPOL specifications), but major efforts are still needed to obtain a true interactive development environment for language definitions.

In this paper, we describe our own contributions to the GIPE project that aim at constructing a "programming environment based on language definitions" as already sketched in [11]. Some ideas on "monolingual programming environments" [12] have also guided our work. We distinguish three phases:

(1) design of an integrated language definition formalism (ASF + SDF);
(2) implementation of a generator that generates environments given a language definition;
(3) design and implementation of an interactive development environment for ASF + SDF.

The latter leads to a *meta-environment* in which language definitions can be edited, checked, and compiled just like programs can be manipulated in a *generated environment* (i.e., an environment obtained by compiling a language definition). Both the generator itself and the meta-environment have been implemented on top of the current Centaur system. Coming back to the issue of closed versus open systems, our system takes a middle position: many mechanisms are built-in and cannot be changed by the user (this leads to an easy-to-use system but probably blocks off certain applications), but there is a well-defined mechanism to connect tools to the generated environments.

The main topics to be discussed are

—interactive editing of modular language definitions with immediate translation of modifications in the language definition to modifications in the programming environment generated for it (this requires in our case, for instance, incremental type checking, incremental scanner and parser generation, and incremental compilation of algebraic specifications);
—treatment of formalisms with variable (i.e., user-definable) syntax.

The plan of the paper is as follows. In Section 2, we give an overview of the features of the formalism ASF + SDF that have influenced the design of the meta-environment. In Section 3, we present the global organization of the ASF + SDF meta-environment. In Section 4, we address the issue of defining the syntax of the equations in modules, and in Section 5 we give a

look inside the generic syntax-directed editor that forms the essential building block in our design. After these preparations, we describe the actual construction of the ASF + SDF meta-environment in Section 6. We describe the implementation techniques needed for the system in Section 7 and conclude the paper with a description of the current state of the implementation as well as a discussion of the relative merits of our approach in Section 8.

2. ASF + SDF

The global design of the meta-environment for ASF + SDF to be discussed in the next section can, to a large extent, be used for a variety of specification formalisms. We make a number of assumptions about specifications and the modules in specifications (e.g., assumptions about the mechanisms for the import and parameterization of modules, for the renaming of names in modules, and assumptions about the specific form of conditional equations). There is, however, one specific feature that has largely determined our design: modules cannot only introduce new functions and define their semantics, but they can introduce new notations for these functions as well. The implications of this feature are far reaching, since one has to provide for the (syntax-directed) editing of specifications with a variable syntax.

Although a detailed understanding of the formalism ASF + SDF is not necessary for understanding the remainder of this paper, a brief sketch of the formalism may help the reader to see the benefits (and associated implementation problems) of user-definable syntax.

ASF + SDF is the result of the marriage of the formalisms ASF (Algebraic Specification Formalism) and SDF (Syntax Definition Formalism). ASF [3] is based on the notion of a module consisting of a signature defining the abstract syntax of functions and a set of conditional equations defining their semantics. Modules can be imported in other modules and can be parameterized. SDF [13] allows the simultaneous definition of concrete (i.e., lexical and context-free) and abstract syntax and implicitly defines a translation from text strings (via their associated parse trees) to abstract syntax trees. The main idea of ASF + SDF [13, 19, 29] is to identify the abstract syntax defined by the signature in an ASF specification with the abstract syntax defined implicitly by an SDF specification, thus yielding a standard mapping from strings to abstract-syntax trees. This gives the possibility to associate semantics with (the tree representation of) strings and to introduce user-defined notation in specifications.

Two (trivial) examples may help to clarify this general description. Figure 1 shows a definition of two modules. Module Booleans defines a sort BOOL, constants true and false, and left-associative operator &. The equations define & as the ordinary *and* operator on Boolean values. Module Naturals defines a sort NAT, constant 0, successor function succ, and infix operator $<$. The equations define $<$ as the ordinary *less than* operator on natural numbers.

This example shows how new syntax rules are introduced in a module (appearing under the heading context-free syntax) and how they can be used in the equations. The result is that, for instance, the equation [B1] can only

```
module Booleans
  exports
    sorts BOOL
    lexical syntax
      [\t\n]                    → LAYOUT
    context-free syntax
      true                      → BOOL
      false                     → BOOL
      BOOL "&" BOOL             → BOOL {left}
  equations
  [B1]  true & true             = true
  [B2]  true & false            = false
  [B3]  false & true            = false
  [B4]  false & true            = false
module Naturals
  imports Booleans
  exports
    sorts NAT
    context-free syntax
      "0"                       → NAT
      succ "(" NAT ")"          → NAT
      NAT " < " NAT             → BOOL
    variables
      N → NAT
      M → NAT
  equations
  [N1]  0 < 0                   = false
  [N2]  succ (N) < 0            = false
  [N3]  0 < succ(N)             = true
  [N4]  succ(N) < succ(M)       = N < M
```

Fig. 1. An ASF + SDF specification of Booleans and Naturals.

be parsed given the preceding syntax definition of the & operator. Since arbitrary context-free grammars can be defined in this way, we cannot give a fixed grammar for each module. Instead, all syntax rules defined in a module (together with all syntax rules defined in imported modules) contribute to the grammar of that particular module (see also Section 4).

Being interested in formal language definitions, we give an example of a (trivial) type-checking problem. Consider the language L of programs of the form

def {*a list of identifiers*} in {*a list of identifiers*}

satisfying the constraint that each identifier appearing in the second list appears in the first list as well. A definition of L is given in Figure 2 and consists of three modules. Module Identifiers defines sorts ID (identifiers) and ID-LIST (lists of identifiers) together with a membership function in. The sort L-PROGRAM introduced in module L-syntax consists of all syntactically correct L-programs. In module L-tc, we define the type-checking function tc[ ] on L-programs that checks the constraint mentioned above.

```
module Identifiers
  imports Booleans
  exports
    sorts ID ID-LIST
    lexical syntax
      [a-z] [a-z0-9]*          → ID
    context-free syntax
      "{" { ID ","}* "}"       → ID-LIST
      ID in ID-LIST            → BOOL
    variables
      Id ['] *          → ID
      Ids ['] *         → {ID ","}*
  equations
  [Id1]  Id in { }       = false
  [Id2]  Id in {Id, Ids}  = true
  [Id3]          Id != Id'
         ==========================
         Id in {Id', Ids} = Id in {Ids}
module L-syntax
  imports Identifiers
  exports
    sorts L-PROGRAM
    context-free syntax
      def ID-LIST in ID-LIST     → L-PROGRAM
module L-tc
  imports L-syntax
  exports
    context-free syntax
      tc "[" L-PROGRAM "]"     → BOOL
  equations
  [Tc1]  tc [ def {Ids} in { } ] = true
  [Tc2]  tc[ def {Ids} in {Id, Ids'} ] =
         Id in {Ids} & tc[def {Ids} in {Ids'} ]
```

Fig. 2. A simple language and its type checker.

3. THE META-ENVIRONMENT

Decomposing large systems into manageable pieces is, of course, an old and well-known engineering technique. Applying modular decomposition techniques to formal language definitions is, however, relatively new. In principle, the benefits to be expected from this approach are the gradual construction of a *library of language definition modules* that can be reused in the formal definition of different languages, e.g., reusing parts of the definition of Fortran expressions in the definition of Pascal (but also see the discussion in Section 8.3).

In this section, we present the architecture of a system for the interactive development of modular language definitions. The main question will be how to give support for the interactive editing of modules and how to update the implementations of these modules automatically after each edit operation.

As already illustrated by the examples in the previous section, we are here

considering a specification formalism in which

—a formal language definition consists of a set of modules;
—a module may import other modules from the language definition;
—each module may define syntax rules as well as semantic rules;
—the notation used in the semantic rules depends on the definition of the syntax rules.

3.1 General Architecture

Figure 3 shows the overall organization of the system. First of all, we make a distinction between the *meta-environment* (i.e., the interactive development environment for constructing language definitions and for generating and testing particular programming environments) and a *generated environment* (i.e., an environment for constructing programs in some programming language L, obtained by compiling a language definition for L in the meta-environment). In the meta-environment one can distinguish:

—a language definition (in ASF + SDF) consisting of a set of modules;
—the environment generator itself, which consists of three components discussed below.

The output of the environment generator is used in conjunction with GSE (Generic Syntax-directed Editor), a generic building block that we use to construct 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 type checking and evaluation. The main inputs to the Generic Syntax-directed Editor are

—a program text P,
—the modules that define the syntax of P,
—connections with external tools.

One language definition can thus result in more than one generated environment by connecting a number of instances of GSE to different sets of external tools. Since 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 grammar rules and equations simultaneously. It may even happen that subsets of these grammar rules and equations are used for *other* purposes in the *same* generated environment.

We will first motivate the architecture sketched in Figure 3 and discuss some details of the environment generator itself. A detailed discussion of GSE is postponed to Section 5.

Our point of departure is a formalism (ASF) in which the operations for module composition (import, export, renaming, parameter binding) are defined in terms of textual expansion: with each module one can associate a new module that does not contain any module composition operations (its so-called *normal form*) by textually expanding each composition operation that appears in the original module. This conceptually simple model is

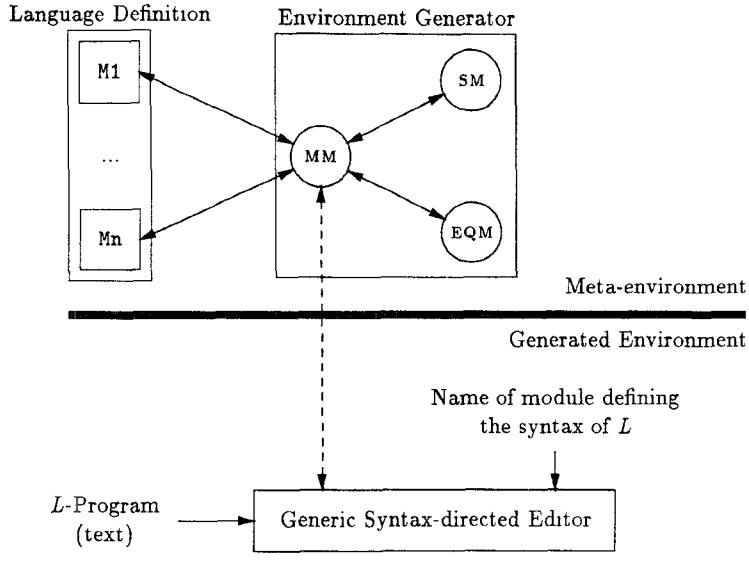

Fig. 3. Global organization.

inadequate as a basis for implementation since the actual copying of modules is not only expensive (both in compilation time and in size of the generated code), but also difficult to extend to incremental compilation of modules.

An ideal implementation model is illustrated in Figure 4. At the specification level, we assume a composition operation + on modules. At the implementation level, we assume both an implementation function $\mathcal{I}$ that maps a module to its implementation and a composition operation $\oplus$ on implementations. Given two modules M_1 and M_2, the following equality has to hold:

$$\mathcal{I}(M_1 + M_2) = \mathcal{I}(M_1) \oplus \mathcal{I}(M_2) \tag{1}$$

From the perspective of interactive editing of modules this property is attractive since we can reuse implementations of unchanged modules. For example, when M_1 is changed into M_1' we can reuse $\mathcal{I}(M_2)$, since

$$\mathcal{I}(M_1' + M_2) = \mathcal{I}(M_1') \oplus \mathcal{I}(M_2) \tag{2}$$

Unfortunately, in practice it is hard to find combinations of $+$, $\oplus$, and $\mathcal{I}$ with this property, since for reasons of efficiency most implementation functions $\mathcal{I}$ will perform global optimizations when constructing $\mathcal{I}(M_1 + M_2)$ which need global information from both M_1 and M_2. The types of modules and the instances of $\oplus$ and $\mathcal{I}$ that we will encounter are summarized in Figure 5. In all these cases, + represents the operator for the textual composition of modules.

We propose therefore the following, alternative, implementation model sketched in Figure 6. Instead of composing implementations we build one implementation for all modules in the specification and make a selection from this global implementation to obtain implementations for individual modules.

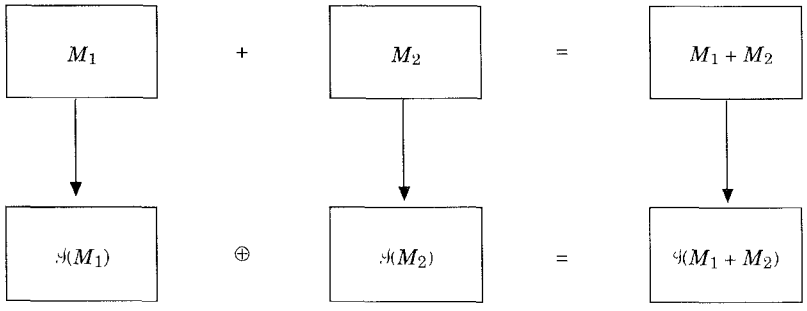

Fig. 4. Ideal implementation scheme.

M	$\mathcal{I}$	$\oplus$
lexical grammar	scanner generator	composition of scanners
context-free grammar	parser generator	composition of parsers
conditional equations	equation compiler	composition of compiled equations

Fig. 5. Types of modules with associated operators $\mathcal{I}$ and $\oplus$.

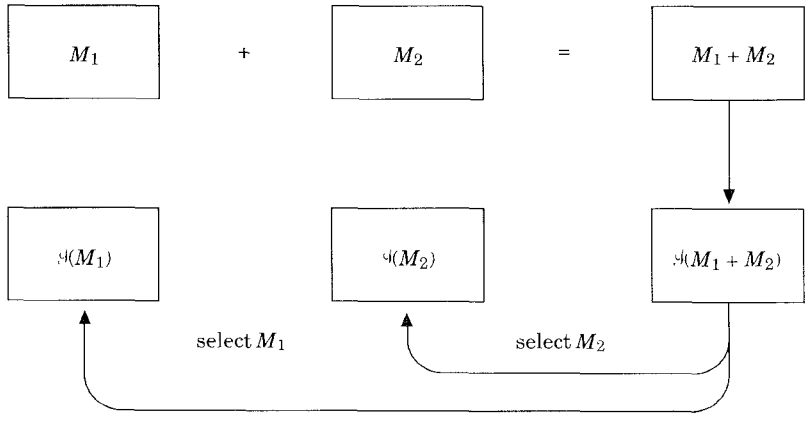

Fig. 6. Actual implementation scheme based on selection.

More specifically, each module in the language definition contains a number of "rules" such as declarations, lexical and context-free grammar rules, and conditional equations. We collect all rules from all modules in a single, global set of rules. Each rule in this global set is tagged with the name of the module in which it was defined. We use these tags to enable or disable individual rules in the global set. Instead of constructing the normal form for each module, we only have to calculate which rules in the global set must be enabled to obtain the same effect as the desired normal form. After selecting certain rules from the global set, these can be used immediately, for instance,

for parsing input sentences according to the selected set of grammar rules, or for rewriting an input term according to the selected set of conditional equations.

Consider, in Figure 7, a sequence of named modules which may contain names of other modules to be imported as well as a number of unspecified "rules" which we denote by lower-case letters. The names declared in an imported module may optionally be renamed before it is imported. The corresponding normal forms are shown in Figure 8 and the corresponding global set of rules in Figure 9. The global set of rules contains the original rules as they appear in the specification together with renamed versions of the rules as needed for the normalization of all the modules in the specification. As an optimization, we could remove from the global set those renamed rules that are identical to the original one, i.e., rules that are not affected by the renaming.

The success of this implementation model is determined by the efficiency of the following operations:

—calculation of the set of rules corresponding to a normal form;
—enabling/disabling rules in the global set;
—selecting parts of the implementation of the rules in the global set for a given set of enabled/disabled rules;
—modifying the global set of rules (and the corresponding implementation) in response to editing operations on the specification.

The viability of this implementation model is further discussed in Section 7.

Returning to the global architecture shown in Figure 3, we distinguish three components in the environment generator that maintain information at a global level:

—The Module Manager (MM) administers the overall modular structure of the language definition. This amounts to maintaining the import relations between modules and keeping track of definition and use of individual rules.
—The Syntax Manager (SM) administers the (lexical and context-free) functions as well as the declarations of priorities (not further discussed here) and variables defined in each module. It also creates and updates the scanners and parsers derived from all modules.
—The Equation Manager (EQM) administers the equations defined in each module together with the rewrite rules that have been derived from them.

The general principle is that the Module Manager manages all modular information and that the Syntax Manager and the Equation Manager can access only the pieces of information that they need to carry out their respective tasks.

Applying this organization to the example specification discussed above, we obtain the situation shown in Figure 7. The Module Manager passes all information related to syntactic issues to the Syntax Manager, which in turn maintains two global sets of rules: lexical rules and context-free rules. All

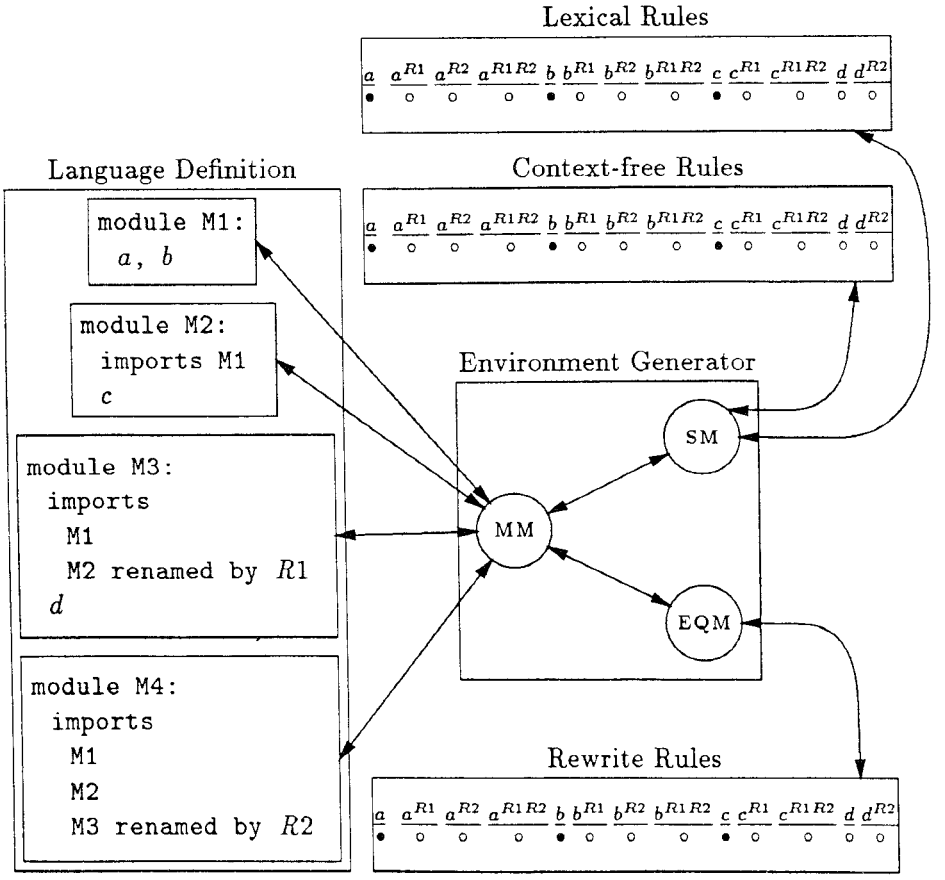

Fig. 7. Processing of a modular specification (module M2 is selected).

module M1: a, b
module M2: a, b, c
module M3: $a, b, a^{R1}, b^{R1}, c^{R1}, d$
module M4: $a, b, c, a^{R2}, b^{R2}, a^{R1R2}, b^{R1R2}, c^{R1R2}, d^{R2}$

Fig. 8 Normal forms of specification in Figure 7.

information related to equations is passed to the Equation Manager, which maintains one global set of rewrite rules derived from the equations.

3.2 Major Components

The Module Manager, the Syntax Manager, and the Equation Manager all adhere to a similar protocol based on the following operations:

add, delete: Add/delete an entity to/from the language definition. Depending on the component, these entities may be a module declaration, a sort

	a	a^{R1}	a^{R2}	a^{R1R2}	b	b^{R1}	b^{R2}	b^{R1R2}	c	c^{R1}	c^{R1R2}	d	d^{R2}
M1	•	○	○	○	•	○	○	○	○	○	○	○	○
M2	•	○	○	○	•	○	○	○	•	○	○	○	○
M3	•	•	○	○	•	•	○	○	○	•	○	•	○
M4	•	○	•	•	•	○	•	•	•	○	•	○	•

Fig. 9. Rules and selections corresponding to specification in Figure 7.

declaration, a lexical function definition (a lexical grammar rule), a context-free function definition (a context-free grammar rule), a priority declaration, an import, a variable declaration, or an equation.

select: Select a module as current module. For the Syntax Manager this implies (1) determining all SDF functions (and their renamed versions) belonging to the normal form of the current module in order to determine the current grammar and (2) selecting the parts of the generated scanner and parser accepting that grammar. For the Equation Manager this implies (1) determining all equations (and their renamed versions) belonging to the normal form of the current module in order to determine the current set of equations and (2) selecting those parts of the compiled term-rewriting system corresponding to the selected set of equations.

parse: Parse a string according to the grammar defined by the currently selected module.

rewrite: Rewrite an abstract syntax tree (usually called "term" in the context of term rewriting) using the rewrite rules derived from the equations defined in the current module.

Most of the operations of the Module Manager depend on the corresponding operations defined in, respectively, the Syntax Manager and the Equation Manager.

4. THE SYNTAX OF EQUATIONS

When constructing the meta-environment based on ASF + SDF, we are confronted with the question of how the syntax of equations can be represented. Defining the syntax of equations in the form of an ordinary module is not only elegant, but it is efficient in terms of implementation effort as well. The syntax of equations should be explicit and localized in a single module, as opposed to, for instance, being dispersed over the implementation of the Module Manager. In this way, it will be easy to change the syntax of equations. This might become relevant when we want to combine SDF with some logical formalism other than ASF.

There are two possible approaches to represent the syntax of equations:

—Use a general grammar to describe the form of equations. In its simplest

form, this grammar would consist of a single rule

$$\langle \text{equation} \rangle ::= \langle \text{term} \rangle \text{"="} \langle \text{term} \rangle$$

where ⟨term⟩ describes all well-formed terms that may appear at the left- or right-hand side of the "=" sign as defined by the SDF section of the current language definition. However, this rule permits equations in which the sorts of both terms are unequal. Therefore these have to be rejected in a separate (very simple) type-checking phase.

—Reject type-incorrect equations already during parsing by adding syntax rules to the grammar for equations of all sorts $S_1, \ldots, S_n$ declared in the language definition. This grammar has the form:

$$\langle \text{equation} \rangle ::= \langle S_1 \rangle \text{"="} \langle S_1 \rangle | \cdots | \langle S_n \rangle \text{"="} \langle S_n \rangle$$

We will now consider the second alternative in more detail. Not only because this is a nonstandard approach deserving some investigation, but also because we can then further exploit the incremental parser generator we already need at the implementation level for handling additions and deletions of grammar rules (see Sections 3 and 7).

4.1 Type Checking Using a Specialized Equation Grammar

Consider an ASF + SDF language definition consisting of the modules $M_1, \ldots, M_n$. In order to define the syntax of equations, this language definition is extended in the following way. First, the module Equations is added that introduces a sort (EQ) for an individual equation and a sort (EQ-SECTION) for a complete equations section. We only discuss a simplified version of the definition of unconditional equations; conditional equations can be defined in a similar way. The definition is

```
module Equations
  exports
    sorts EQ EQ-SECTION
    context-free syntax
      EQ* → EQ-SECTION
```

Next, we generate for each module M_i in the language definition a module EQ-M_i that consists of three parts:

(1) For all exported sorts $S_1, \ldots, S_k$ declared in M_i we generate declarations for exported functions of the form S_j"="$S_j \rightarrow$ EQ;.

(2) For all hidden sorts $T_1, \ldots, T_l$ declared in M_i we generate declarations for hidden functions of the form T_j"="$T_j \rightarrow$ EQ;.

(3) For all modules $N_1, \ldots, N_m$ imported by M_i we generate imports of the "equation version" of each module N_j. In case of importing and renaming a module, we simply rename the equation version of N_j and import that renamed module. If the module has no imports, only an import for the module Equations is imported.

The result is as follows:

```
module EQ-M_l
  exports
    context-free syntax
      S_1"="S_1 → EQ
      ...
      S_k"="S_k → EQ
  hiddens
    context-free syntax
      T_1"="T_1 → EQ
      ...
      T_l"="T_l → EQ
  imports
      EQ-N_1 ... EQ-N_m
```

Parsing an equation in module M_l can now be done in the context of the dynamically generated module EQ-M_l.

4.2 Example of a Specialized Equation Grammar

Consider the specification of Booleans and Naturals given earlier in Figure 1 (Section 2). Using the scheme described in the previous paragraph, this specification will be extended with the following modules (apart from the module Equations given earlier):

```
module EQ-Booleans
  exports
    context-free syntax
      BOOL"="BOOL → EQ
  imports Equations
module EQ-Naturals
  exports
    context-free syntax
      NAT"="NAT → EQ
  imports EQ-Booleans
```

An equation like 0 < succ(0) = succ(0) < succ(succ(0)) that could legally appear in module Naturals, can be parsed using EQ-Naturals. More interestingly, an equation like true = succ(0) would be syntactically incorrect.

5. LOOKING INSIDE GSE

The Generic Syntax-directed Editor (GSE) is a generic building block providing the following functionality:

—syntax-directed editing of strings (programs) in a given language L;
—connecting "external tools" operating on the L program in the editor. As we will see, some of these "external tools" will be derived from the language definition itself (e.g., type checker, evaluator, code generator).

5.1 Syntax-Directed Editing

GSE aims at integrating text-oriented editing and structure-oriented editing as smoothly as possible. See [24], [25], and [31] for a detailed description

and Section 8 for some remarks on the relative merits of these two editing paradigms.

By syntax-directed navigation (or just by pointing) the user can position a *focus* on a part of the program being edited. The text outside the focus is always syntactically correct, but the contents of the focus can be modified by conventional text-editing operations. The user can move the focus to another part of the program, with or without parsing the text in the current focus. When, in the former case, syntax errors are found, the new focus will simply cover both the old (erroneous) part of the program and the new part the user wants to move to. In addition, GSE provides commands for the template-driven creation of program fragments.

Other aspects of GSE worth mentioning are the following:

—Unlike most other syntax-directed editors, GSE *does not* use pretty printing to recreate the text of programs from its internal-tree representation. Instead, a two-way mapping is maintained between the text as typed in by the user and the tree representation. In this way, the user has complete control over the layout of the program (but can of course request to reformat parts of it), and the well-known problem of pretty printing comments can be solved in a straightforward manner.

—To support editing in the meta-environment (see Section 6), GSE can handle the case that modifications are made to the syntax of the input language L currently in use. After such a modification, it should be verified that the current program in the editor is still a valid L program. The naive implementation we currently use is to completely (re)parse the program.

—The possibility to extend GSE's user interface to connect external tools as described below.

5.2 Attaching External Tools to the Editor

The formal definition of a language may contain rules specifying certain operations on programs such as type checking and evaluation. After compilation of the specification this leads to a number of functions that can operate on programs. From the viewpoint of the editor these functions form "external tools," and the question now arises as to how they can be attached to an instance of GSE. The following points should be considered (see also Figure 10):

Activate external tool. Add a button or menu entry to GSE's standard user-interface which activates the external tool. Activation of the external tool takes the form of a possibly parameterized function call. Some external tools (like, for instance, an incremental type checker) need to be called *implicitly* whenever the program in the editor is changed. For this we need a notion of change propagation, to be discussed in Section 5.5.

Make information available to external tool. Depending on the information required by the external tool, information like the focus or the whole

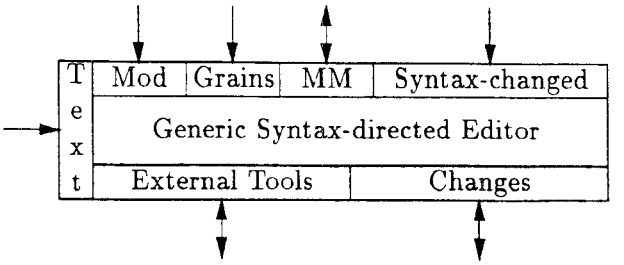

Fig. 10. Generic Syntax-directed Editor (GSE) with its parameters.

program in the current (or another named) GSE instance can be passed as parameters.

Retrieve results of the external tool. There are several possible destinations for the output of the external tool such as the focus or the whole program in the current or another, named or a newly created, GSE instance.

The essential design decision here is that the form of the input and output data of the external tool will be defined in ASF + SDF and that they are therefore treated as "input language" and "output language" of the external tool.

5.3 Customization

Having identified the need to add external tools to GSE's syntax-directed user interface, the question now arises how to describe such customizations.

A first approach we tried was to define a collection of built-in data types for describing the user interface and add these in the form of a library of modules to ASF + SDF. However, the number of new notions required as well as the interfacing problems involved led us to the conclusion that we could better design a little, special-purpose, language.

We have developed such a language and have called it SEAL [23], which stands for Semantics-directed Environment Adaptation Language. Its main purpose is to "seal" together the user interface and semantic functions described in the language definition. A SEAL script describes which user interface elements should be added to which editor. For instance, the typical definition of a *button* contains the following information:

—A *name* used as label to identify the button in the user interface.

—*Enabling conditions* determining when the button can be activated or not. Enabling conditions may be purely syntactic (e.g., the focus is currently positioned at a statement) but may also be semantic in nature (e.g., the focus is currently positioned at a type-incorrect expression). In the latter case, arbitrary functions may be called to compute the enabling condition.

—*Actions* to be performed when the button is activated by the user. Actions can be composed of the following primitives:
  —focus expressions, to extract and replace subterms from the current (or a named other) editor instance;
  —application of functions as described in the language definition;
  —creation of new editors or dialogues.

As a result, the specification writer can fully control *when* the button may be activated, *how* it collects input parameters for calling some function defined in the specification, and *how and where* it leaves the result of this function. This scheme allows the description of buttons like a "type check button" that presents a list of error messages in a new window, or a "transformation button" that performs a local transformation on the expression currently in focus.

In the next section, we give some typical examples how SEAL can be used to customize generated environments.

5.4 Using GSE as a Building Block

Typical examples of the use of GSE and SEAL are shown in Figures 11 and 12. In Figure 11, the language definition consists of a single module M, and we construct an environment for editing and evaluating terms in M. The "external tool" connected to GSE rewrites the current tree using the equations from module M. This connection is established by the following SEAL description:

```
button reduce
{
  root := M : root
}
```

First of all, a button labeled reduce is added to GSE's user interface. When pressing this button the following happens:

—Using the select function of the Module Manager, module M is selected as current module (this is described by "M :").
—Using the rewrite function of the Module Manager, the whole tree in the current editor is reduced using the equations in module M ("M : root").
—The whole tree in the current editor is replaced by the result of this reduction ("root := M : root").

In Figure 12, the language definition consists of three modules: L-SYN (defining the syntax of language L), L-TC (defining the type checking of L-programs; L-TC imports L-SYN), and L-EV (defining the evaluation of L-programs; it also imports L-SYN). In this case, we construct an environment for editing, type checking, and evaluating L-programs. The buttons check and execute are implemented using the functions tc and eval defined in, respectively, L-TC and L-EV. When pressing, for instance, the check button, the following happens:

—In the context of module L-TC, the function tc is applied to the whole program in the current editor. The result (an expression of sort ERRORS) is assigned to the variable Errors.

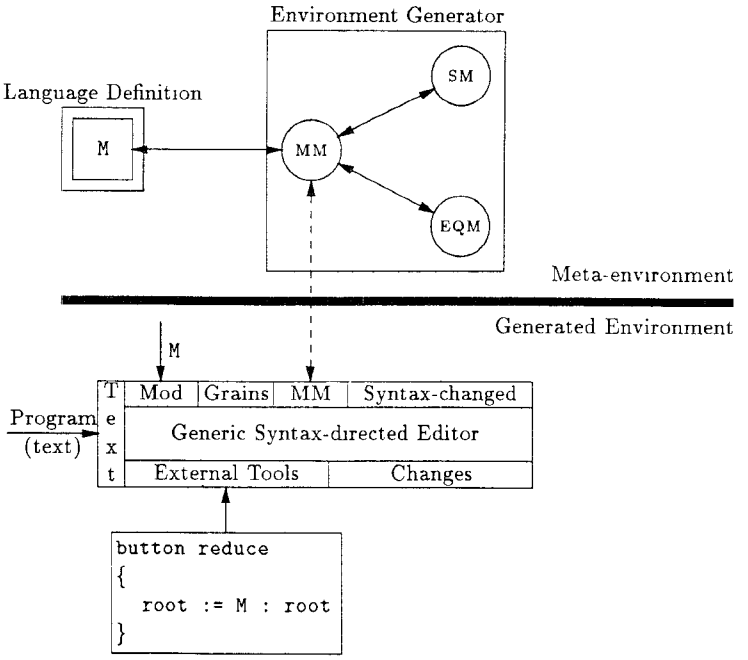

Fig. 11. Generated environment for evaluating terms.

—A new GSE instance is created with the title Type Checking Errors and as contents the value of the variable Errors.

The overall effect is thus that on pressing the check button, a new window pops up containing error messages. This example illustrates the point that the output of each tool is considered to be an expression in some "language," in this case the language described by the sort ERRORS.

5.5 Propagating Changes

GSE is parameterized with a function Changes that communicates changes to the attached tools. In principle, there are two possibilities for choosing the granularity of this communication:

—Call Changes after each modification to the program.

—Call Changes only after modifications that exceed certain "grain sizes" that are given as a parameter of the editor.

In the first case, Changes has to infer whether additional actions are needed, whereas in the second case this can be done by the editor in a more generic way.

In general, there will be a mismatch between the size of a change made during editing and the size of the changes the external tool can cope with. For instance, if the external tool can handle changes of the size of statements (in the context of editing some programming language) how do we process

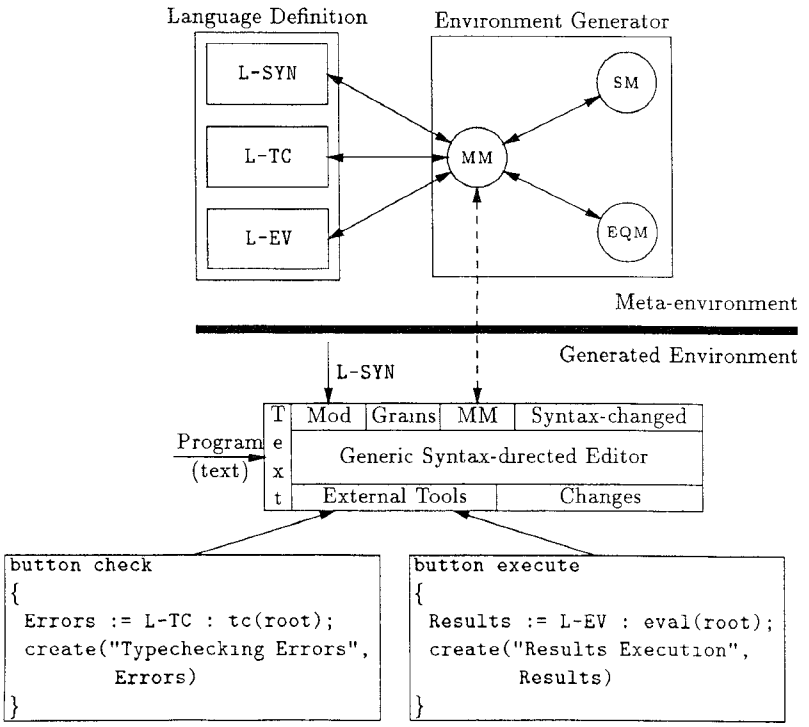

Fig. 12. Generated environment for type-checking/evaluating L-programs.

changes to parts of a statement such as the condition in an if statement? The approach we have chosen is to determine the smallest grain enclosing a modification automatically and call the external tool for it. Modifications to program fragments that are larger than the grains provided by the external tool are processed by calculating the difference between the old and the new fragment and calling the external tool for a minimal number of grains that cover the difference. See [18, chapter 7] for a complete description of the algorithm performing these grain calculations.

In the current version of GSE, definitions of grain sizes are used internally in the implementation to optimize the incremental behavior of the components of the meta-environment itself. They are, however, not yet available to the language designer. As a result, the standard way of propagating changes is to call the external tool after each modification of the program.

6. EDITING IN THE META-ENVIRONMENT

How can we use generated editing environments to edit ASF + SDF specifications? To answer this question we have to define the complete syntax of ASF + SDF specifications. This can be done in the following way:

—To each specification we add, implicitly, a fixed module called SDF, which defines the syntax of the SDF part of each module.

—To each specification we add the module Equations defining the syntax of equations as described in Section 4.
—To each module M_i we add a module EQ-M_i, defining the contributions of module M_i to the syntax of equations.

Editing a module M_i in the specification now amounts to creating *two* editors: one for the SDF part of the module (GSE_1, which uses the fixed module SDF as grammar definition) and one for the equations part (GSE_2, which uses the generated module EQ-M_i as grammar definition). This is shown in Figure 13. Some comments on this figure are appropriate:

—The grain size for the processing of changes to the SDF part is determined by a list of sorts given to GSE_1. This list contains a sort name for each entity for which the Syntax Manager provides add/delete operations.
—The Changes function associated with GSE_1 will use the Syntax Manager for actually performing the changes to the SDF part of a module. It will also call Syntax changed of GSE_2 after each modification to the SDF part of the module.
—The grain size for GSE_2 is determined by a list only containing the sort EQ, i.e., only changes at the level of complete equations are considered as changes. This corresponds precisely to the add/delete operations provided by the Equation Manager.
—The Changes function associated with GSE_2 will use the Equation Manager for actually performing the changes to the equations apart of the module.
—We have left unspecified which operations are performed on, respectively, the SDF part and the equations part of the module. Typical examples are type checking and compiling.

7. IMPLEMENTATION TECHNIQUES

In Section 3.1 we have presented an implementation model for modular specifications in which all "rules" appearing in modules are collected in one global set together with a mechanism to select individual rules from this set. Finding an efficient implementation method for this model is, of course, essential. Although a general framework for describing such a method is still lacking, two experiments have been performed and documented that demonstrate the feasibility of the approach.

One experiment [22] concerns the case that the rules in each module are regular expressions to be compiled into a deterministic finite automaton. The key idea is to construct a *single* automaton for all regular expressions in all modules. The selection operation that enables or disables certain regular expressions is implemented by enabling or disabling the corresponding transitions in the automaton. The resulting Modular Scanner Generator uses techniques for *lazy* and *incremental* program generation [15, 16]: parts of the finite automaton are only constructed when they are needed, and most parts not affected by the addition or deletion of a regular expression will be reused. In the same spirit, the enabling or disabling of transitions is only

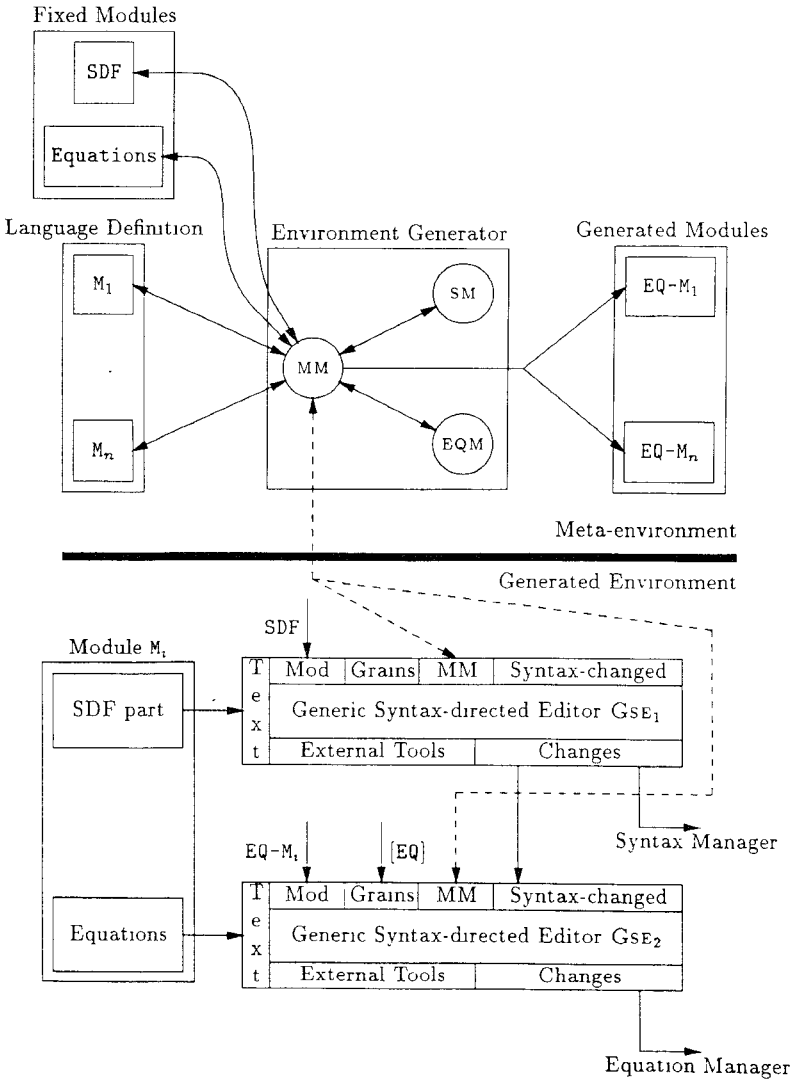

Fig. 13 Editing language definition module M_i

done when needed. In [33], an experiment is described that uses finite automata for the matching of left-hand sides of rules and applies selection techniques similar to the ones described here.

The other experiment [26] concerns modular context-free grammars, and the "rules" to be considered are syntax rules. Key idea is, again, to construct a single parse table for all syntax rules in all modules and to implement the enabling or disabling of a syntax rule by enabling or disabling the corresponding transitions in the parse table. The resulting Modular Parser

Generator also uses lazy and incremental techniques and extends the notion of incremental parser generation described in [17].

Measurements show that in the above cases the selection operation is very fast and that the overall performance of the generated programs is hardly influenced by the introduction of a selection mechanism for individual rules. We have therefore concluded that an efficient implementation along the lines of our model exists, and we have based the implementation of our whole system on it. Experience with the system confirms the above initial observations.

8. CONCLUDING REMARKS

8.1 Current State of the Implementation

The meta-environment for generating programming environments as presented in this paper has been completely implemented. It supports the interactive development of ASF + SDF specifications by providing syntax-directed editing of specifications and immediate translation of modifications made to the ASF + SDF specification to modifications in the programming environment generated for it. The generated environment is immediately available and can be used for experimentation. The current meta-environment consists of the following parts:

—A Module Manager consisting of a nucleus providing the operations as described in this paper, except that the current implementation does not yet support renaming and parameterization of modules. On top of this nucleus, a simple user interface has been built yielding an interactive development environment for language definitions written in ASF + SDF.

—A Syntax Manager based on the Modular Scanner Generator and Modular Parser Generator discussed in the previous section.

—An Equation Manager that does a certain amount of preprocessing on the equations but is essentially an interpreter of conditional rewrite rules. Its speed is in the order of 1,000 rewriting steps per second on a SparcStation 2.

—The Generic Syntax-directed Editor: It supports integrated text-oriented and structure-oriented editing, but does not yet provide the complete functionality as described in Section 5. In particular, the determination of the grain size is currently implemented as part of the Module Manager and is only available for built-in components of the meta-environment.

—A first implementation of SEAL has been completed. It supports user-defined enabling/disabling conditions for buttons and menu entries, navigation commands, and primitives for creating interactive dialogues.

8.2 Work in Progress

In GSE we have tried to integrate text-oriented and structure-oriented editing as much as possible. However, we have not invested in the implementation of

yet another text editor. Our experience now shows that the limited amount of text-editing primitives we provide makes GSE a less-than-convenient editor. This leads us to the conclusion that structure-oriented editing should be seen as a *complement* rather than as a *replacement* of a good text editor. To validate this observation, we are now removing all text-editing primitives from GSE and will start using Emacs as a "text-editing server" instead.

Although GSE is not based on pretty printing, we still need it for the on-demand reformatting of programs and for printing new values resulting from evaluation. In the current system, a fixed pretty printer is derived automatically from the grammar rules in the specification. This "built-in" solution is not always appropriate and will have to be extended to satisfy all formatting needs.

Using techniques described in [28], we are now able to derive *incremental* implementations from ASF + SDF specifications. The implementation of incremental functions is currently being optimized and will soon be integrated in the meta-environment.

An Equation Compiler is near completion that compiles conditional equations to C programs. Initial measurements indicate a speedup factor of 50–150 over the current, already quite fast, Equation Manager. This opens the perspective that compiled ASF + SDF specifications run at a speed comparable to that of C programs.

A notion of "origins" has been defined [30] that establishes a relation between terms as they appear during rewriting and the initial term. A first implementation of origins exists, which will be integrated into the system in order to provide a uniform mechanism for relating error messages to source code, for debugging, and for animation of execution.

We plan to describe and generate all user interface components of the meta-environment by means of SEAL scripts.

8.3 Discussion

A full evaluation of the ASF + SDF meta-environment has to wait until further extensions and optimizations of its implementation have been completed and until more extensive experience with its use has been gathered. Nevertheless, some remarks on the design are in order.

8.3.1 *Problems and Open Questions*

—Modular decomposition of language definitions could, in principle, lead to libraries of reusable language definition modules. Unfortunately, our experience so far indicates that we need very expressive operations for module composition to reflect the trivial and detailed differences in syntax and semantics of "similar" notions in different languages. This is clearly an area for further research.

—The system proposed here will be faced with a serious version management problem: after changing a language definition there may still be programs around that conform to the old definition.

—It is not yet clear whether the proposed implementation model based on selection will scale up to industrial-size applications.

—Not much experience exists with the use of specification formalisms with user-definable syntax. In principle, freedom of notation seems to be a desirable property, but it may very well turn out that this freedom has to be controlled in some way for the sake of readability and reusability of specifications.

—Experience with a specialized equation grammar as discussed in Section 4.1 reveals that this method works satisfactorily, but that it introduces additional requirements for the error-reporting capabilities of the parser since type errors in equations will be reported as *syntax* errors.

8.3.2 *Merits of the* ASF + SDF *Meta-environment.* 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 implementation. For instance, the presence of a parser generator is completely invisible to the user. As a result, the system is also accessible to "naive" users who have no previous experience with the use of tools like scanner and parser generators. Important factors are: (1) the abstract syntax and the pretty printer for the language are derived automatically from the language definition; (2) after parsing, abstract 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* the components of the generated environment.

The generality of the syntax definition mechanism provided by the formalism ASF + SDF together with the new, but well-understood, techniques used for their implementation form an improvement over the syntax definition facilities in comparable systems [9, 32].

The use of two coupled instances of GSE for editing languages definitions in the meta-environment is an interesting case of reusing existing components. As a result, both the meta-environment and generated environments will benefit from future improvements in GSE.

The similarity between the meta-environment and generated environments leads to a situation where features considered desirable in the meta-environment may have unexpected applications in generated environments (and vice versa). This may lead to interesting generalizations.

ACKNOWLEDGMENTS

Constructing a programming environment based on language definitions is a common, long-term, goal shared with Jan Heering. We have had numerous discussions about the desirability, implications, and possible realization of such a system. The specific design and implementation of the meta-environment presented here has emerged from numerous discussions with Huub Bakker (user interfaces, GSE-Emacs coupling), Jan Bergstra (ASF), Arie van Deursen (origins, Module Manager, error reporting), Niek van Diepen, Casper Dik (Equation Manager), Hans van Dijk (Generic Syntax-directed

Editor), Jan Heering (general design, ASF, SDF), Paul Hendriks (ASF, SDF, Module Manager), Jasper Kamperman (Equation Compiler), Wilco Koorn (Generic Syntax-directed Editor, SEAL), Monique Logger (Generic Syntax-directed Editor), Emma van der Meulen (incremental evaluation), Jan Rekers (SDF, Syntax Manager), Frank Tip (Equation Debugger, origins), Ard Verhoog, and Pum Walters (Equation Manager, Equation Compiler).

Last but not least, comments made by *users* of the ASF + SDF meta-environment have led to several improvements in the system. Jan Heering, Paul Hendriks, Wilco Koorn, and Emma van der Meulen commented on drafts of this paper.

REFERENCES

1. BAHLKE, R., AND SNELTING, G. The PSG system. From formal language definitions to interactive programming environments. *ACM Trans. Program. Lang. Syst. 8*, 4 (1986), 547–576.
2. BALLANCE, R. A., GRAHAM, S. L., AND VAN DE VANTER, M. L. The Pan language-based editing system. *ACM Trans. Softw. Eng. Meth. 1*, 1 (1992), 95–127.
3. BERGSTRA, J. A., HEERING, J., AND KLINT, P., EDS. *Algebraic Specification*. ACM Press Frontier Series, New York, 1989.
4. BORRAS, P., CLÉMENT, D., DESPEYROUX, T., INCERPI, J., KAHN, G., LANG, B., AND PASCUAL, V. Centaur: The system. In the *3rd Annual Symposium on Software Development Environments (SIGSOFT'88)* (Boston, 1988). ACM, New York.
5. DESPEYROUX, T Executable specification of static semantics. In *Semantics of Data Types*, vol. 173. Lecture Notes in Computer Science. Springer-Verlag, New York, 1984, 215–233.
6. DONZEAU-GOUGE, V., HUET, G., KAHN, G., AND LANG, B. Programming environments based on structured editors the Mentor experience. In *Interactive Programming Environments*. McGraw-Hill, New York, 1984, 128–140.
7. ENDRES, R., AND SCHNEIDER, M. The GRASPIN software engineering environment. In *ESPRIT '88: Putting the Technology to Use*. North-Holland, Amsterdam, 1988, 349–364.
8. ESPRIT (An overview of Genesis. Project 1222 (GENESIS), 1987. Deliverable 12Y3
9. FUTATSUGI, K., GOGUEN, J. A., JOUANNAUD, J.-P., AND MESEGUER, J. Principles of OBJ2. In *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, 52–66.
10. HABERMANN, A. N., AND NOTKIN, D. Gandalf: Software development environments. *IEEE Trans. Softw Eng. 12*, 12 (1986), 1117–1127.
11. HEERING, J. Een programmeeromgeving gebaseerd op taaldefinities. In *Colloquium Programmeeromgevingen*, vol. 30. Mathematical Centre Syllabus, 1983, 69–81. In Dutch.
12. HEERING, J., AND KLINT, P. Towards monolingual programming environments. *ACM Trans. Program. Lang Syst. 7*, 2 (1985), 183–213.
13. HEERING, J., HENDRIKS, P. R. H., KLINT, P., AND REKERS, J. The syntax definition formalism SDF—reference manual. *SIGPLAN Not. 24*, 11 (1989), 43–75.
14. HEERING, J., KAHN, G., KLINT, P., AND LANG, B. Generation of interactive programming environments. In *ESPRIT '85: Status Report of Continuing Work*. North-Holland, Amsterdam, 1986, 467–477.
15. HEERING, J., KLINT, P., AND REKERS, J. Incremental generation of lexical scanners. *ACM Trans. Program. Lang. Syst. 14*, 4 (Oct. 1992), 471–520.
16. HEERING, J., KLINT, P., AND REKERS, J. Lazy and incremental program generation. *ACM Trans. Program. Lang. Syst.* To be published.
17. HEERING, J , KLINT, P., AND REKERS, J. Incremental generation of parsers. *IEEE Trans. Softw. Eng. 16*, 2 (1990), 1344–1351.
18. HENDRIKS, P. R. H. Implementation of modular algebraic specifications. Ph.D. thesis, University of Amsterdam, 1991.

19. HENDRIKS, P. R. H. Lists and associative functions in algebraic specifications—semantics and implementation. Rep. CS-R8908, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1989.
20. KAHN, G. Natural semantics. In the *4th Annual Symposium on Theoretical Aspects of Computer Science*, vol. 247. Lecture Notes in Computer Science. Springer-Verlag, New York, 1987, 22–39.
21. KAHN, G., LANG, B., MÉLÈSE, B., AND MORCOS, E. Metal: A formalism to specify formalisms. *Sci. Comput. Program. 3*, 2 (1983), 151–188.
22. KLINT, P. Lazy scanner generation for modular regular grammars. Tech. Rep. CS-R9158, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1991.
23. KOORN, J. W. C. Connecting semantic tools to a syntax-directed user-interface. Tech. Rep., Programming Research Group, University of Amsterdam, 1992.
24. KOORN, J. W. C. GSE: A generic text and structure editor. Rep. P9202, University of Amsterdam, 1992.
25. LOGGER, M. H. An integrated text and syntax-directed editor. Rep. CS-R8820, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1988.
26. REKERS, J. Modular parser generation. Rep. CS-R8933, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1989.
27. REPS, T., AND TEITELBAUM, T. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, New York, 1989.
28. VAN DER MEULEN, E. A. Deriving incremental implementations from algebraic specifications. Rep. CS-R9072, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1990.
29. VAN DER MEULEN, E. A. Algebraic specification of a compiler for a language with pointers. Rep. CS-R8848, Centrum voor Wiskunde en Infomatica (CWI), Amsterdam, 1988.
30. VAN DEURSEN, A., KLINT, P., AND TIP, F. Origin tracking. Rep. CS-R9230, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1992.
31. VAN DIJK, M. H. H., AND KOORN, J. W. C. GSE, a generic syntax-directed editor. Rep. CS-R9045, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1990.
32. VOISIN, F. CIGALE: A tool for interactive grammar construction and expression parsing. *Sci. Comput. Program. 7*, 1 (1986), 61–86.
33. WALTERS, H. R. On equal terms, implementing algebraic specifications. Ph.D. thesis, University of Amsterdam, 1991.

Received December 1991; revised July and November 1992; accepted November 1992