Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

P. Klint

A meta-environment for generating programming environments

Computer Science/Department of Software Technology Report CS-R9064 November

A Meta-Environment for Generating Programming Environments

P. Klint
Department of Software Technology, Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
and
Programming Research Group, University of Amsterdam
P.O. Box 41882, 1009 DB Amsterdam, The Netherlands

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, but 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 very soon become the crucial factor determining the 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.

1989 CR Categories: 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.

1985 Mathematics Subject Classification: 68N15 [**Software**]: Programming Languages; 68N20 [**Software**]: Compilers and generators.

Key Words & Phrases: concrete and abstract syntax, user-definable syntax, programming language semantics, algebraic specification, language definition formalism, programming environment generation, meta-environment, incremental program generation.

Note: Partial support received from the European Communities under ESPRIT project 2177 (Generation of Interactive Programming Environments II—GIPE II).

To appear in: J.A.Bergstra & L. Feijs (eds), *Proceedings of the METEOR workshop on Methods based on formal specifications*, Mierlo, September 1989, to appear in Springer Lecture Notes in Computer Science.

Report CS-R9064
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

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 [DHKL84], PSG [BS86], Synthesizer Generator [RT89], Gandalf [HN86], GIPE [HKKL85], Genesis [GENESIS87], and Graspin [ES88]). A programming environment is here understood as a coherent set of interactive tools such as syntax-directed editors, debuggers, interpreters, code generators, and prettyprinters to be used during the construction of texts in the desired language. This approach has been used to generate environments for languages in different areas such as programming, formal specification, text formatting, and proof construction. 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 non-incremental 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.).

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, but it is difficult or even impossible to use them in other areas. Therefore, one should strive for systems with more open architectures consisting of cooperating sets of replaceable components. In this way one can obtain as much generality and flexibility as possible. It may, of course, turn out that very general systems are difficult to use in *every* application area.

The Centaur system [BCDIKLP88] is an outcome of 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;
- 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 TYPOL [Des84, Kah87], SDF [HK89a], METAL [KLMM83], 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 [Hee83]. Some ideas on "mono-lingual programming environments" [HK85] have also guided our work. We distinguish three phases:

- Design of an integrated language definition formalism (ASF+SDF);
- Implementation of a generator that generates environments given a language definition;
- 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.
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 typechecking, 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 of the meta-environment 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 [BHK89] 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 [HK89a, HHKR89] 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 [HHKR89, HK89b, Hen89, vdM88] 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 $\wedge$. The equations define $\wedge$ 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.

```
 1. module Booleans
 2.    exports
 3.      sorts BOOL
 4.      lexical syntax
 5.        [ \t\n]            -> LAYOUT
 6.      context-free  syntax
 7.        true               -> BOOL
 8.        false              -> BOOL
 9.        BOOL "∧" BOOL      -> BOOL {left}
10.    equations
11.    [1]  true ∧true        = true
12.    [2]  true ∧false       = false
13.    [3]  false ∧true       = false
14.    [4]  false ∧false      = false

15. module Naturals
16.    exports
17.      sorts NAT
18.      context-free  syntax
19.        0             -> NAT
20.        succ NAT      -> NAT
21.        NAT "<" NAT   -> BOOL
22.    imports Booleans
23.    variables
24.        N -> NAT
25.        M -> NAT
26.    equations
27.    [1]  0 < 0             = false
28.    [2]  succ N < 0        = false
29.    [3]  0 < succ N        = true
30.    [4]  succ N < succ M   = N < M
```

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

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 in line 11 can only be parsed given the syntax definition in line 9. 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) typechecking 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 ∈. The sort `L-PROGRAM` introduced in module `L-syntax` consists of all syntactically correct *L*-programs. In module `L-tc`, we define the typechecking function `tc[]` on *L*-programs that checks the constraint mentioned above.

```
module Identifiers
   exports
     sorts ID, ID-LIST
      lexical syntax
         [a-z] [a-z0-9]*   -> ID
         [ \t\n]           -> LAYOUT
       context-free syntax
          "{" ID* "}"      -> ID-LIST
          ID "∈" ID-LIST   -> BOOL
   imports Booleans   -- as defined in Figure 1
   variables
     Id [']*     -> ID
     Ids         -> ID*
   equations
   [1] Id ∈ {}         = false
   [2] Id ∈ {Id Ids} = true
                     Id ≠ Id'
   [3]  ----------------------------------
          Id ∈ {Id' Ids}  =  Id ∈ {Ids}

module L-syntax
   exports
     sorts L-PROGRAM
      context-free syntax
       def ID-LIST in ID-LIST -> L-PROGRAM
   imports Identifiers

module L-tc
   exports
     context-free syntax
      tc "[" L-PROGRAM "]"    -> BOOL
   imports L-syntax
   variables
     Id   -> ID
     Ids  -> ID*
     Defs -> ID*
   equations
   [1]  tc [ def {Defs} in {} ] = true
        Id ∈ Defs = true, tc [ def {Defs} in {Ids} ] = true
   [2]  -------------------------------------------------------
                  tc[ def {Defs} in {Id Ids} ] = true
                     Id ∈ Defs = false
   [3]  --------------------------------------
         tc[ def {Defs} in {Id Ids} ] = false
```

Figure 2. A simple language and its typechecker.

The points to be emphasized in these examples are:

- 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. GLOBAL ORGANIZATION OF A META-ENVIRONMENT BASED ON ASF+SDF

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 typechecking and evaluation. The main inputs to the Generic Syntax-directed Editor are:

- a program text *P*;
- the module that defines the syntax of *P*;
- connections with external tools.

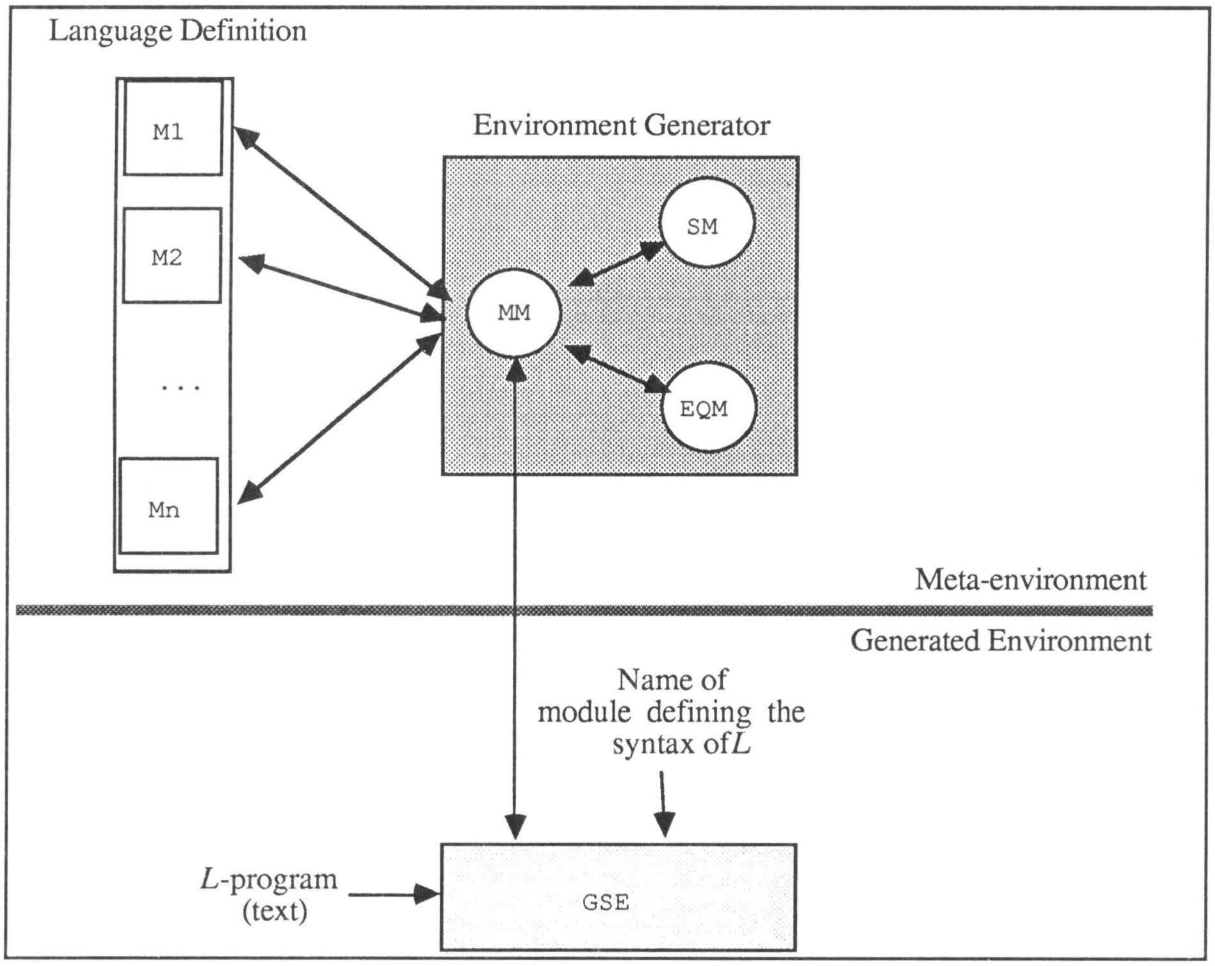


Figure 3. Global organization.

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.

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. As previous research has shown [Hen88], this conceptually simple method is 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 separate compilation of modules.

We propose the following, alternative, implementation model. Each module in the language definition contains a number of "rules" such as declarations, 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, one only has to calculate which rules in the global set have to 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. 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.

Consider, in Figure 4, 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 5 and the corresponding global set of rules in Figure 6. 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, one 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.

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

1. 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.
2. The Syntax Manager (SM) administers the (lexical and context-free) functions as well as the declarations of priorities and variables defined in each module. The Syntax Manager also creates and updates the scanners and parsers derived from all modules.
3. The Equation Manager (EQM) administers the equations defined in each module together with the rewrite rules that have been derived from them.

```
module M1 begin
             a, b
          end M1
module M2 begin
             imports M1
             c
          end M2
module M3 begin
             imports M1,
                     M2 renamed by R1
             d
          end M3
module M4 begin
             imports M1,
                     M2,
                     M3 renamed by R2
          end M4
```

Figure 4. A modular specification

module M1 **begin**
    a, b
  end M1
module M2 **begin**
    a, b, c
  end M2
module M3 **begin**
    a, b, a^{R1}, b^{R1}, c^{R1}, d
  end M3
module M4 **begin**
    a, b, c, a^{R2}, b^{R2}, a^{R1R2}, b^{R1R2}, c^{R1R2}, d^{R2}
  end M4

Figure 5. Normal forms of the modules in Figure 4.

	a	a^{R1}	a^{R2}	a^{R1R2}	b	b^{R1}	b^{R2}	b^{R1R2}	c	c^{R1}	c^{R1R2}	d	d^{R2}
M1	x	o	o	o	x	o	o	o	o	o	o	o	o
M2	x	o	o	o	x	o	o	o	x	o	o	o	o
M3	x	x	o	o	x	x	o	o	o	x	o	x	o
M4	x	o	x	x	x	o	x	x	x	o	x	o	x

Figure 6. Global set of rules and selections corresponding to example in Figure 4.

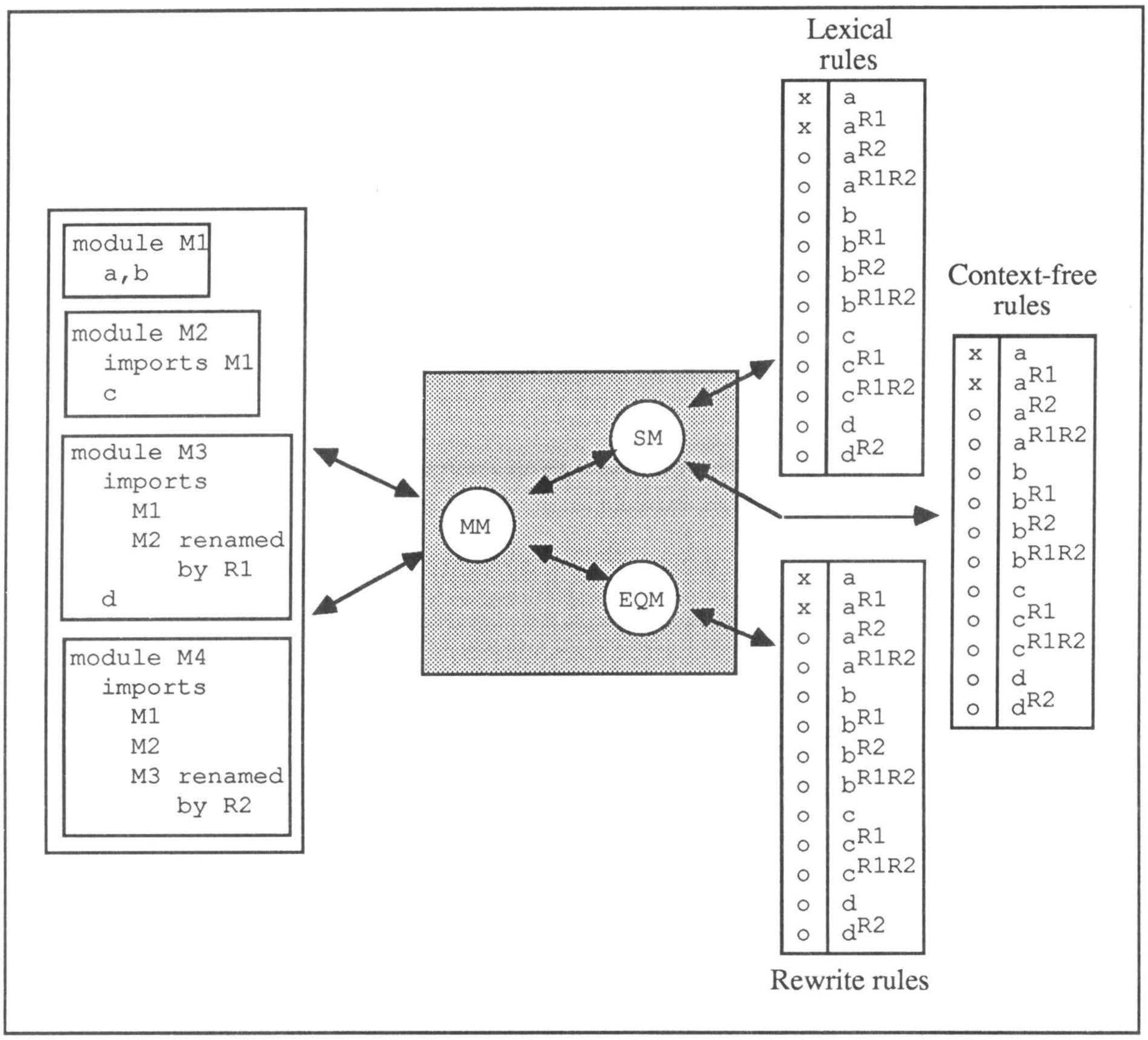


Figure 7. Processing the specification of Figure 4.

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 given earlier in Figure 4, 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 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

Next, we give a description of all operations provided by the Syntax Manager, the Equation Manager, and the Module Manager.

3.2.1. The Module manager (MM)

The Module Manager provides operations for adding and deleting modules and parts of modules as well as for parsing strings and evaluating abstract syntax trees:

`add, del:`
: Add/delete a module to/from the language definition; add/delete one of the following entities to/from a module: sort declaration, lexical function definition, context-free function definition, priority declaration, import, variable declaration, or equation.

`select:`
: Select a module as current module.

`parse:`
: Parse a string in the context of the current module; the result is an abstract syntax tree.

`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.

Many of these operations depend on the corresponding operations defined in, respectively, the Syntax Manager and the Equation Manager (see below).

3.2.2. The Syntax Manager (SM)

The Syntax Manager provides operations for adding and deleting parts of the SDF-section of a language definition, for selecting a module, and for parsing strings:

`add, del:`
: Add/delete one of the following entities to/from a given module: lexical function definition, context-free function definition, variable declaration, priority declaration, renaming, import, or equation.

`select:`
: Select a module as current module. All SDF functions (and their renamed versions) belonging to the normal form of the current module define its grammar. Select the parts of the generated scanner and parser accepting that grammar.

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

3.2.3. The Equation Manager (EQM)

The Equation Manager provides operations for adding and deleting equations to/from a module, for selecting a module, and for evaluating terms:

`add, del:`
: Add/delete a signature element or an equation to/from a given module.

`select:`
: Select a module as current module. All equations (and their renamed versions) belonging to the normal form of the current module define its set of equations. Select those parts of the compiled term rewriting system corresponding to the selected set of equations.

`rewrite:`
: Rewrite a term according to the currently selected module.

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 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

```
        <equation> ::= <term> "=" <term>,
```

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. Unfortunately, this rule permits equations in which the sorts of both terms are unequal. Therefore these have to be rejected in a separate type checking phase.

- Reject type incorrect equations already during parsing by adding syntax rules to the grammar for equations of all sorts s_1, ..., s_n declared in the language definition. This grammar has the form:

```
<equation> ::= <S1> "=" <S1> | ... | <Sn> "=" <Sn>
```

We will now consider the second alternative in more detail.

4.1. Typechecking equations by means of a specialized equation grammar

Consider an ASF+SDF language definition consisting of the modules M_1,..., M_n (see Figure 8). 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 for an individual equation and a sort 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 $EQ\text{-}M_i$ that consists of three parts:

- For all exported sorts S_1,...,S_k declared in M_i we generate declarations for exported functions of the form `Sj "=" Sj -> EQ;`
- For all hidden sorts T_1,...,T_l declared in M_i we generate declarations for hidden functions of the form `Tj "=" Tj -> EQ;`
- For all modules N_1,...,N_m imported by M_i we generate imports of the "equation-version" of each module N_j. If the module has no imports, only an import for the module `Equations` is imported.

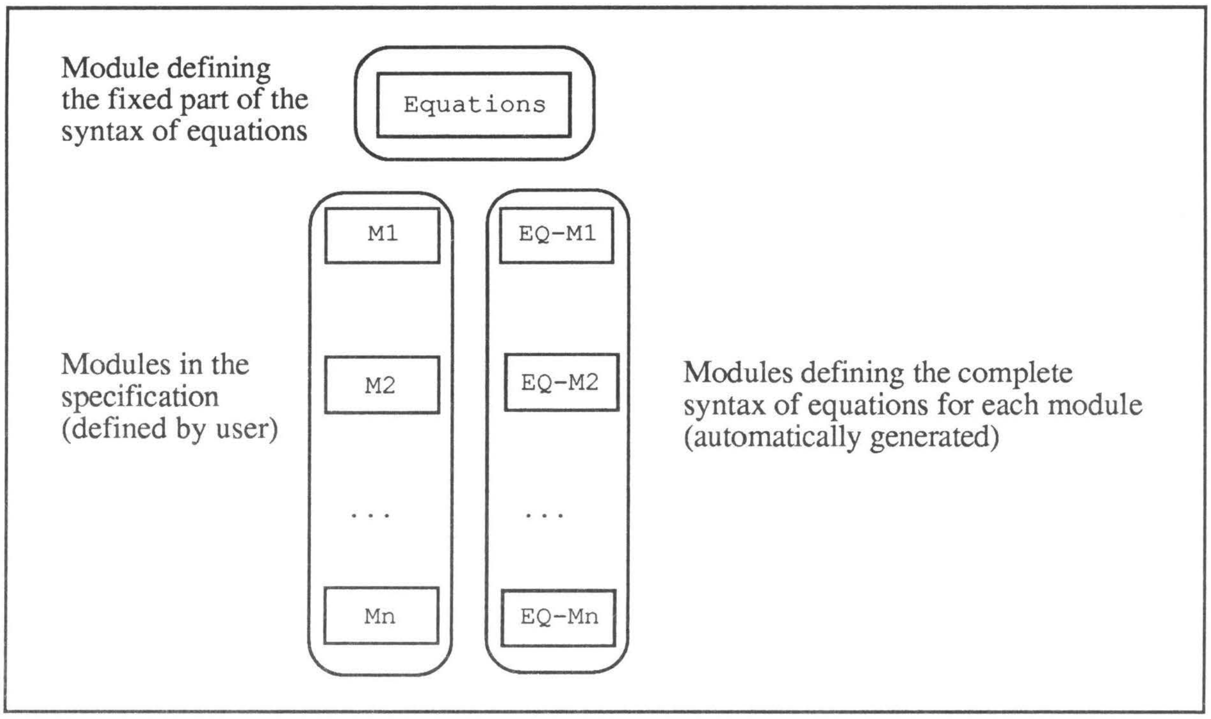
The result is as follows:

```
module EQ-Mi
    exports
       context-free syntax
          S1 "=" S1 -> EQ
          ...
          Sk "=" Sk -> EQ
    hiddens
       context-free syntax
          T1 "=" T1 -> EQ
          ...
          Tl "=" Tl -> EQ
    imports
          EQ-N1
          ...
          EQ-Nm
```

Parsing an equation in module M_i can now be done in the context of the dynamically generated module $EQ\text{-}M_i$.

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 modules `Equation` and `Equations` given earlier):



Module defining the fixed part of the syntax of equations
Equations
M1
EQ-M1
Modules in the specification (defined by user)
M2
EQ-M2
Modules defining the complete syntax of equations for each module (automatically generated)
...
...
Mn
EQ-Mn

Figure 8. Definition of the syntax of equations.

```
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 THE GENERIC SYNTAX-DIRECTED EDITOR

The Generic Syntax-directed Editor (GSE) provides the following functions:

- Syntax-directed editing of strings (programs) in a given language *L*.
- Activation of "external tools" operating on the *L*-program in the editor. As we will see some of "external tools" will be derived from the language definition itself (e.g., typechecker or evaluator).
- Display the output of these tools.
- Informing external tools connected to the editor about changes made during editing.
- Adjust the internal state of the editor after a modification to the syntax of language *L*.

We will now briefly discuss each of them.

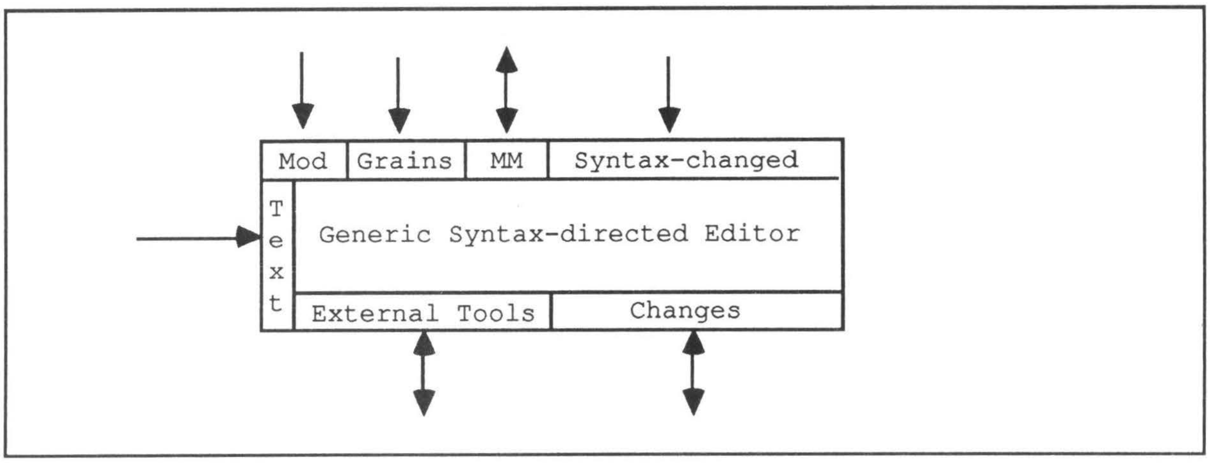


Figure 9. Generic Syntax-directed Editor (GSE) with its parameters.

5.1. Syntax-directed editing

As experience shows, pure syntax-directed editing is not very convenient. In many cases, a user wants to perform editing operations that are text-oriented rather than structure-oriented in nature. To overcome this problem, GSE aims at integrating text-oriented editing and structure-oriented editing as smoothly as possible. By syntax-directed navigation (or just by pointing) the user can position a *focus* on a part of the program being edited. The contents of the focus can be modified by conventional text-editing operations. When the user wants to move the focus to another part of the program, its text is parsed. If syntax errors are found they must be corrected before the focus can be moved. See [DK90, Koo90, Log88] for a description of GSE.

From the perspective of the meta-environment, the parsing of programs can be implemented using the `parse` function of the Module Manager.

5.2. Management of changes

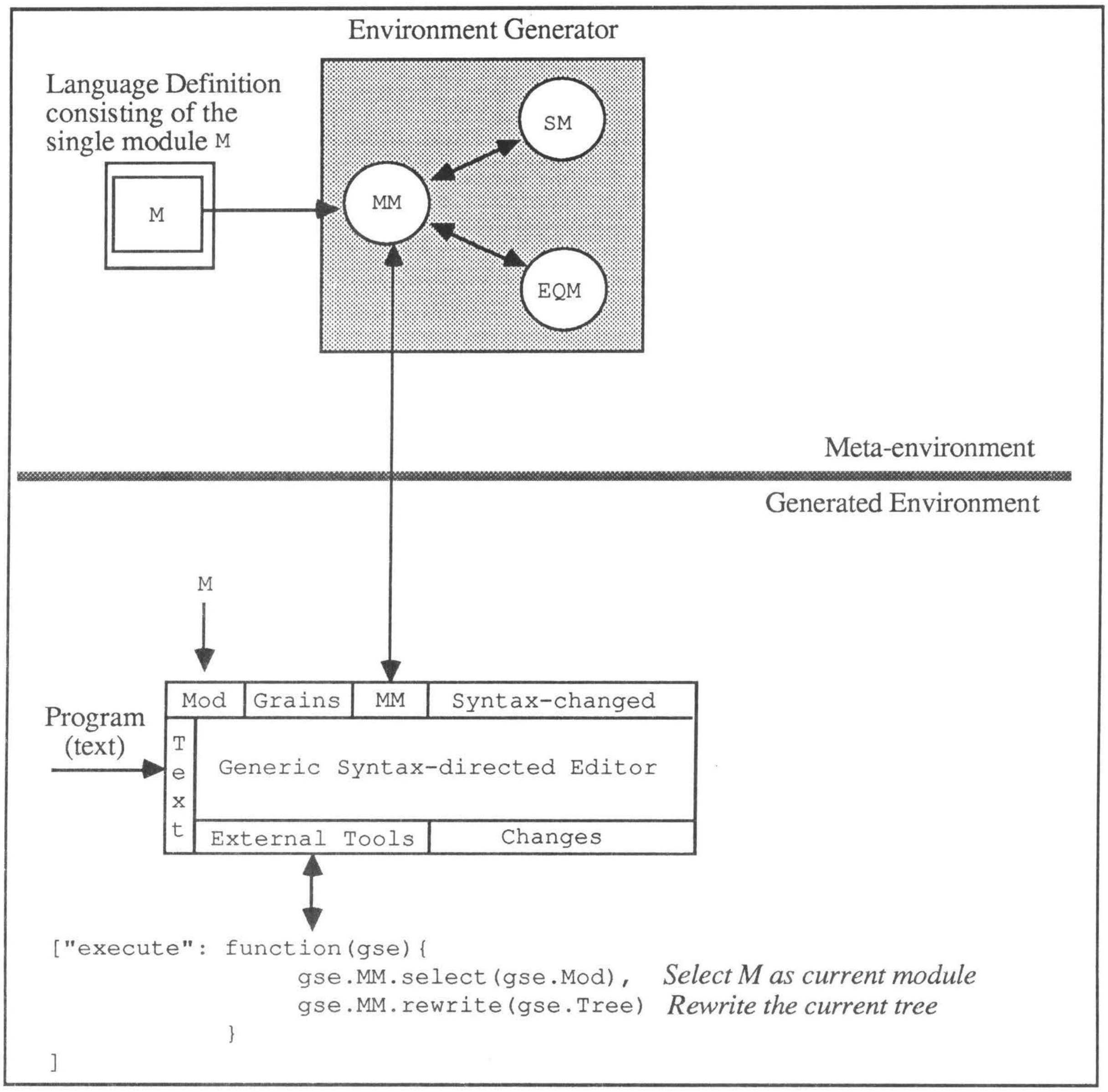
Making changes to programs is an essential task of the editor. It depends on the environment in which the editor is being used whether additional processing is required after a change. Assuming the editor is parameterized with a function `Changes` that communicates changes to the environment, there are two possibilities for choosing the granularity of the communication:

- The function `Changes` is called after each modification to the program.
- It is only called 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 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.

In the design as presented here we only allow for a single definition of grain sizes for each instance of GSE. This implies that all attached external tools will have to provide operations for the same set of grains. In the future it may turn out to be desirable to allow different grain sizes for each external tool attached.



```
["execute": function(gse){
                  gse.MM.select(gse.Mod),    Select M as current module
                  gse.MM.rewrite(gse.Tree)   Rewrite the current tree
            }
]
```

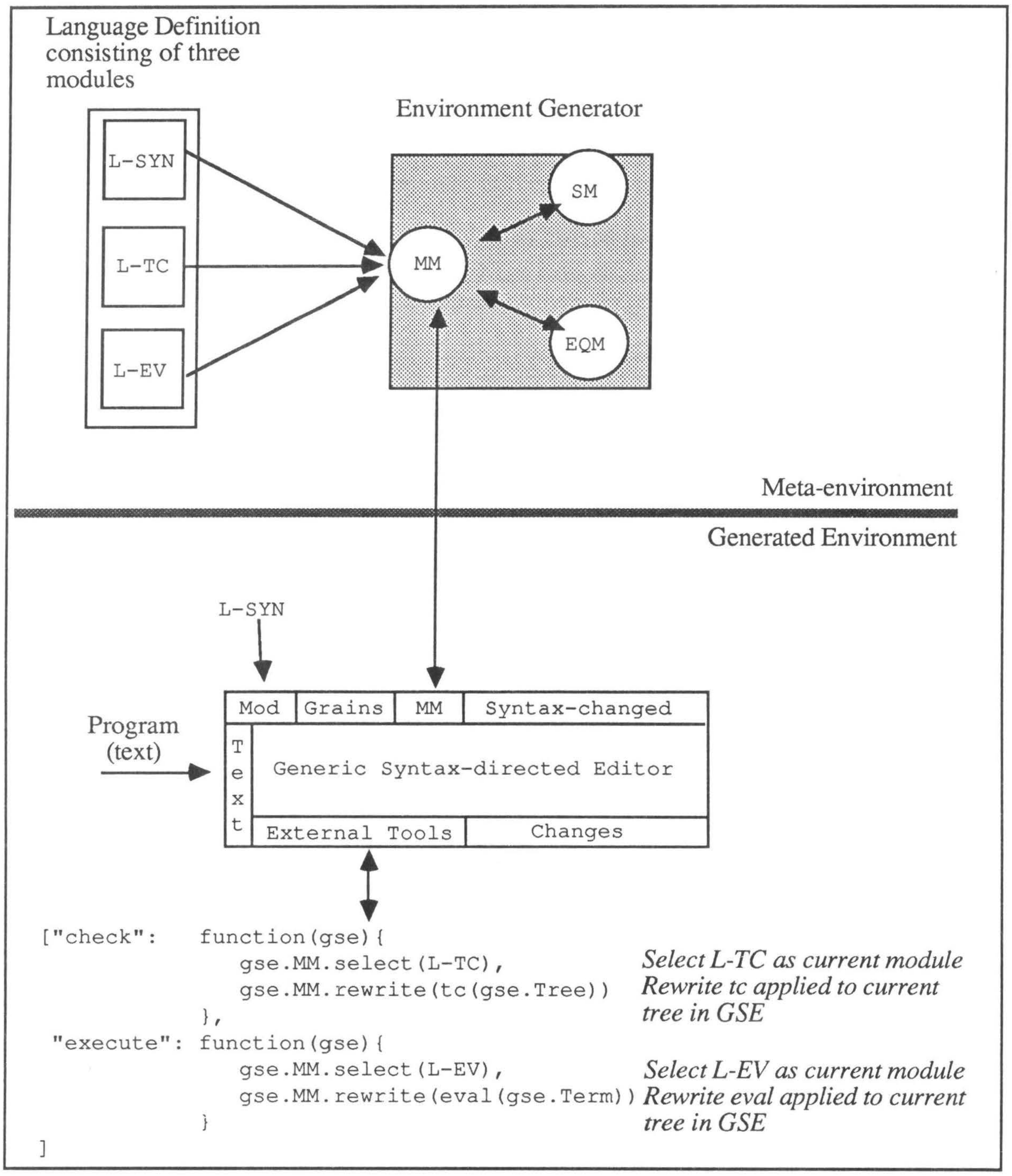
Figure 10. A generated environment for evaluating terms.

5.3. Attaching external tools to the editor

The formal definition of a language may contain rules specifying certain operations on programs such as typechecking 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 GSE. The following points should be considered:

(a) Activation of the external tool.
(b) Communication of information from within the editor to the external tool.
(c) Communication of the output of the external tool back to the editor.

Point (a) can be solved by including a list of (command,function)-pairs in the language definition and passing this list as a parameter to the editor. The commands are placed in the command menu of the editor and selection of a certain entry from the menu will result in a call to the associated function. All functions have the same calling interface: they have the editor



```
["check":    function(gse){
                gse.MM.select(L-TC),           Select L-TC as current module
                gse.MM.rewrite(tc(gse.Tree))   Rewrite tc applied to current
             },                                tree in GSE
 "execute": function(gse){
                gse.MM.select(L-EV),           Select L-EV as current module
                gse.MM.rewrite(eval(gse.Term)) Rewrite eval applied to current
             }                                 tree in GSE
]
```

Figure 11. A generated environment for editing, typechecking and evaluating L-programs.

from which they are being called as single argument. Note that *automatic* activation of external processors (as, for instance, needed for incremental typechecking) can be implemented by means of the `Changes` function discussed in the previous section.

Point (b) is solved by providing operations on the editor that return (parts of) the internal state of the editor, such as, the current program, the current focus, etc.

Point (c) can only be solved if all external operations return their output in a fixed format. An obvious choice is a list of (error-message, subtrees)-pairs, to be interpreted as a list of all erroneous subtrees with their associated error-messages.

5.4. Syntax modifications

After a modification to the syntax of the input language *L* of the editor, it should be verified that the current program in the editor is still a valid *L*-program. A naive implementation will completely (re)parse the program. This facility is needed to support editing in the meta-environment (see Section 6).

5.5. Major functions of GSE

The above discussion can be summarized in the following list of operations provided by GSE (see also Figure 9):

`GSE`:
  Construct a new instance of GSE given:
    a Module Manager,
    a module name (defining the input syntax, i.e., the syntax of the texts to be edited),
    a definition of the grain sizes for which `Changes` should be called)
    a function `Changes`,
    a list of (command, function)-pairs defining the communication with external tools

`Focus, Tree, MM, Mod`:
  Return status information such as the current focus (`Focus`), the current program (`Tree`), the Module Manager used (`MM`), the module defining the input language (`Mod`), etc.

`Up, Down, Replace, Search`, ..., :
  Perform editing operations.

`Syntax-changed`:
  Signal a modification of the input syntax and adjust the internal state of the editor accordingly.

Typical examples of the use of GSE are shown in Figures 10 and 11. In Figure 10, 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 (command, function)-pair:

```
"execute" :  function(gse){
                    gse.MM.select(gse.Mod),
                    gse.MM.rewrite(gse.Tree)
             }
```

The `execute` command is added to the command repertoire of GSE, for instance, by adding an appropriately labelled button to its user-interface. On activation of the new command, the above function is called. It is a nameless function with the instance of GSE from which it is being called as single parameter. Through this parameter—and all operations provided by GSE—relevant information about the internal state of the editor and of its parameters becomes accessible. In the body of the function `gse.MM` determines the Module Manager associated with this instance of GSE. Using this Module Manager, the current input language of the editor (`gse.Mod`) is selected as current module. Next, the current abstract syntax tree in the editor (`gse.Tree`) is rewritten in the context of the selected module using the `rewrite` function provided by the Module Manager.

In Figure 11, the language definition consists of three modules: `L-SYN` (defining the syntax of language L), `L-TC` (defining the typechecking 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, typechecking and evaluating L programs. The commands `check` and `eval` are implemented using the functions `tc` and `eval` defined in, respectively, `L-TC` and `L-EV`.

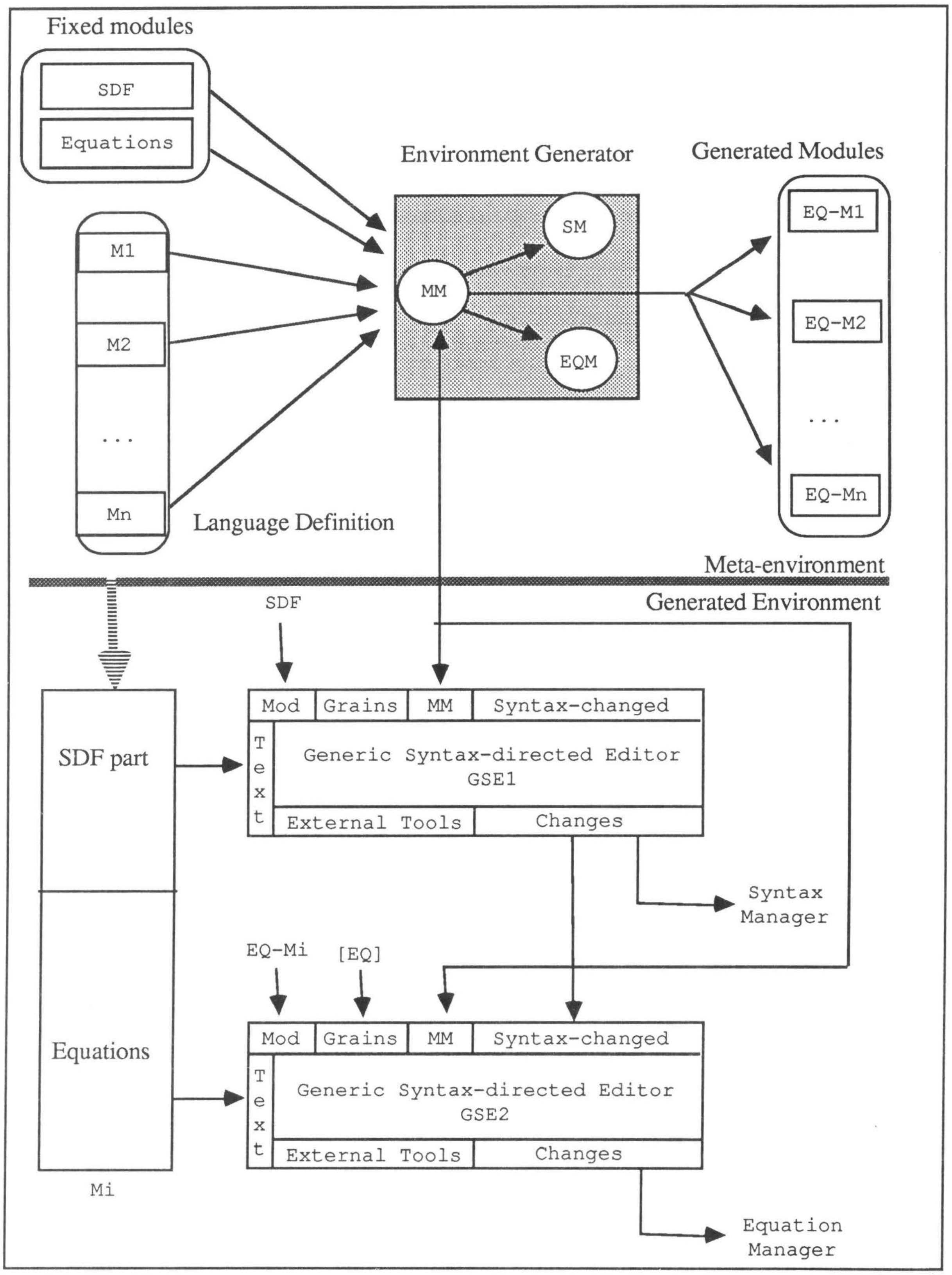


Figure 12. Editing a language definition module.

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` we add a module `EQ-M`, defining the contributions of module `M` to the syntax of equations.

Editing a module in the specification now amounts to creating *two* editors: one for the SDF part of the module (`GSE1`) and one for the equations part (`GSE2`). This is shown in Figure 12. 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 `GSE1`. This list contains a sort name for each entity for which the Syntax Manager provides add/delete operations.
- The `Changes` function associated with `GSE1` will use the Syntax Manager for actually performing the changes to the SDF part of a module. It will also call `GSE2.Syntax-changed` after each modification to the SDF part of the module.
- The grain size for `GSE2` 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 `GSE2` 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: typechecking 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 enable or disable 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 that demonstrate the feasibility of the approach.

One experiment [Kli89] 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 [HKR87a, HKR87b]: 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 done when needed.

The other experiment [Rek89] 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 [HKR90].

Measurements show that in both cases the selection operation is very fast and that the overall performance of the generated programs (i.e., scanner and parser) is hardly influenced by the introduction of an enabling/disabling mechanism for individual rules. We may therefore conclude that an efficient implementation along the lines of our model exists in at least these two specific cases.

In this context, no experience yet exists with the compilation of algebraic specifications to rewrite rules. One of the directions we have investigated is to use finite automata for the matching of left-hand sides of rules and to apply techniques similar to the ones used in the modular scanner generator in this case as well [Wal90].

8. CONCLUDING REMARKS

8.1. Current state of the implementation

A first version of the meta-environment for generating programming environments as presented in this paper has been 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.
- The Modular Scanner Generator and Modular Parser Generator discussed in the previous section are operational and have been used to implement the Syntax Manager.
- A moderately efficient Equation Manager has been implemented. Work on building a more efficient one is in progress.
- The Generic Syntax-directed Editor has been implemented. 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 the connection of external tools has only been implemented in a rudimentary fashion.

Some major implementation problems still have to be solved such as the generic treatment of the output of external tools, i.e., prettyprinting arbitrary values resulting from the evaluation of an external tool, or treating error messages produced by a typechecker in a nice fashion.

8.2. Discussion

Although a full evaluation of the proposed meta-environment has to await completion of its implementation as well as experience with its use, some remarks on the design are in order. We foresee the following problems and open questions:

- 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 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.
- We are not yet able to derive *incremental* implementations from ASF+SDF specifications, but intend to do so using techniques described in [vdM90].

We see the following merits in the proposed system:

- 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.

- The generality of the syntax definition mechanism provided by 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 [FGJM85, Voi86].
- 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 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.

ACKNOWLEDGEMENTS

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 Casper Dik (EQM), Hans van Dijk (GSE), Jan Heering, Paul Hendriks (MM), Wilco Koorn (GSE), Emma van der Meulen, Jan Rekers (SM), and Pum Walters (EQM). Jan Heering, Paul Hendriks and Emma van der Meulen commented on drafts of this paper.

REFERENCES

[BCDIKLP88] P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang & V. Pascual, "Centaur: the system", *Proceedings of the ACM SIGSOFT/SIGPLAN Conference on Practical Software Development Environments*, 1988, pp. 14-24.

[BHK89] J.A. Bergstra, J. Heering & P. Klint (eds.), *Algebraic Specification*, ACM Press in co-operation with Addison-Wesley, 1989.

[BS86] R. Bahlke & G. Snelting, "The PSG system: from formal language definitions to interactive programming environments", *ACM Transactions on Programming Languages and Systems*, Vol. 8, Number 4, 1986, pp. 547-576.

[CH89] D. Clément & L. Hascoët, "Centaur Paths: a structure to designate subtrees", *CENTAUR User's Manual, Version 0.9*, INRIA, 1989.

[DHKL84] V. Donzeau-Gouge, G. Huet, G. Kahn & B. Lang, "Programming environments based on structured editors: the Mentor experience" in D.R. Barstow, H. E. Shrobe & E. Sandewall (eds.), *Interactive Programming Environments*, McGraw-Hill, 1984, pp. 128-140.

[DK90] M.H.H. van Dijk & J.W.C. Koorn, "GSE: a generic syntax-directed editor", Report CS-R9045, Centre for Mathematics and Computer Science, Amsterdam, 1990.

[Des84] T. Despeyroux, "Executable specification of semantics", in *Semantics of Data Types,* G. Kahn, D.B. MacQueen & G. Plotkin (eds.), *Lecture Notes in Computer Science*, Vol. 173, Springer-Verlag, 1984, pp. 215-233.

[ES88] R. Endres & M. Schneider, "The GRASPIN Software Engineering Environment", in *ESPRIT '88: Putting the Technology to Use*, North-Holland, 1988, pp. 349-364.

[FGJM85] K. Futatsugi, J. A. Goguen, J.-P. Jouannaud, and J. Meseguer, "Principles of OBJ2", in *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, ACM, 1985, pp. 52-66.

[GENESIS87] "An Overview of Genesis", ESPRIT Project 1222 (GENESIS), Deliverable 12Y3, 1987.

[HHKR89] J. Heering, P.R.H. Hendriks, P. Klint & J. Rekers, "The syntax definition formalism SDF—reference manual", *SIGPLAN Notices*, Vol. 14 (11), pp. 43-75 (November 1989).

[Hee83] J. Heering, "Een programmeeromgeving gebaseerd op taaldefinities" (in dutch), in J. Heering & P. Klint (eds.), *Colloquium Programmeeromgevingen*, Mathematical Centre Syllabus 30, 1983, pp. 69-81.

[HK85] J. Heering & P. Klint, "Towards monolingual programming environments", *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 2, April, 1985, pp. 183-213.

[HK89a] J.Heering & P. Klint, "The syntax definition formalism SDF", in [BHK89, Chapter 6]. Also in *ESPRIT '86: Results and Achievements*, North-Holland, 1987, pp. 619-630.

[HK89b] J. Heering & P. Klint, "PICO revisited", in [BHK89, Chapter 9]. Also in *ESPRIT '88: Putting the Technology to Use*, North-Holland, 1988, pp. 365-379.

[HKKL85] J. Heering, G. Kahn, P. Klint & B. Lang, "Generation of interactive programming environments", in *ESPRIT '85: Status Report of Continuing Work*, Part I, North-Holland, 1986, pp. 467-477.

[HKR87a] J. Heering, P. Klint & J. Rekers, "Principles of lazy and incremental program generation", Report CS-R8749, Centre for Mathematics and Computer Science, Amsterdam, 1987.

[HKR87b] J. Heering, P. Klint & J. Rekers, "Incremental generation of lexical scanners", Report CS-R8761, Centre for Mathematics and Computer Science, Amsterdam, 1987 (accepted for publication in *ACM Transactions on Programming Languages and Systems*).

[HKR90] J.Heering, P.Klint & J.Rekers, "Incremental generation of parsers", *IEEE Transactions on Software Engineering*, SE-16, 11 (November 1990); Also in *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, (June 1989), pp. 179-191.

[Hen88] P.R.H. Hendriks, "ASF system user's guide", Report CS-R8823, Centre for Mathematics and Computer Science, Amsterdam, 1988.

[Hen89] P.R.H. Hendriks, "Type-checking Mini-ML", in [BHK89, Chapter 7]. Abbreviated version in *Proceedings of CSN87: Computing Science in the Netherlands*, SION, 1987, pp. 21-38.

[HN86] A.N. Habermann & D. Notkin, "Gandalf: software development environments", *IEEE Transactions on Software Engineering*, Vol. 12, 1986, pp.1117-1127.

[KLMM83] G. Kahn, B. Lang, B. Mélèse & E. Morcos, "METAL: a formalism to specify formalisms", *Science of Computer Programming*, Vol. 3, 1983, pp. 151-188.

[Kah87] G. Kahn, "Natural semantics", in *Fourth Annual Symposium on Theoretical Aspects of Computer Science*, ed. F.J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, Lecture Notes in Computer Science, Vol. 247, Springer-Verlag, 1987, pp. 22-39.

[Kli89] P. Klint, "Scanner generation for modular regular grammars", in *Liber Amicorum, J.W. de Bakker, 25 jaar Semantiek*, Centre for Mathematics and Computer Science, Amsterdam, 1989, pp. 291-305.

[Koo90] J.W.C. Koorn, "GSE: A generic text and structure editor", Programming Research Group, University of Amsterdam, to appear, 1990.

[Log88] M. Logger, "An integrated text and syntax-directed editor", Report CS-R8820, Centre for Mathematics and Computer Science, Amsterdam, 1988.

[vdM88] E.A. van der Meulen, "Algebraic specification of a compiler for a language with pointers", Report CS-R8848, Centre for Mathematics and Computer Science, Amsterdam, 1988.

[vdM90] E.A. van der Meulen, "Deriving incremental implementations from algebraic specifications", Centre for Mathematics and Computer Science, Amsterdam, to appear, 1990.

[RT89] T. Reps & T. Teitelbaum, *The Synthesizer Generator: a System for Constructing Language-based Editors,* Springer-Verlag, 1989.

[Rek89] J. Rekers, "Modular parser generation", Report, Centre for Mathematics and Computer Science, Amsterdam, CS-R8933, 1989.

[Voi86] F. Voisin, "Cigale: a tool for interactive grammar construction and expression parsing", *Science of Computer Programming*, Vol 7., 1986, pp. 61-86.

[Wal90] H.R. Walters, "Using text scanners for tree matching", Programming Research Group, University of Amsterdam, to appear, 1990.