The Equation Debugger

Frank Tip
Programming Research Group
University of Amsterdam
April 1991

Contents

1. Introduction 2
2. The formalism ASF + SDF 4
   - 2.1 Introduction 4
   - 2.2 ASF + SDF 5
   - 2.3 An example: the specification of arithmetic expressions 6
   - 2.4 Basic Modules 6
   - 2.5 The syntax of expressions 10
   - 2.6 A typechecker for expressions 11
   - 2.7 An evaluator for expressions 13
3. The ASF + SDF system 14
   - 3.1 Overview 14
   - 3.2 The ASF + SDF meta-system 15
   - 3.3 The Generic Syntax-directed Editor GSE 16
   - 3.4 Term-Editors 18
   - 3.5 Module-Editors 19
   - 3.6 The internal structure of the ASF + SDF system 20
4. Term-rewriting in the ASF + SDF system 22
5. The functionality of the Equation Debugger 26
   - 5.1 Introduction 26
   - 5.2 Objectives and Design Issues 26
   - 5.3 The user-interface of EDB 27
   - 5.4 The interface between EDB and EQM 33
   - 5.5 An example: tracing the evaluation of a term 36
6. Debugging term rewriting systems compared with debugging conventional programs 42
7. Concluding Remarks 44

Acknowledgements 44
References 45

- cum laude
- 5½ maand
- allerlei baantjes
- cursus Lotus/Dbase
- gisteren nieuwe baan

- observer
- lessen t.a.v. EQM

1. Introduction

The subject of this report is a debugger for term rewriting systems. Differences and analogies between debugging term rewriting systems, and debugging programming languages will be analysed. The debugger we present here is called the Equation Debugger (EDB). Before we can describe the global design and implementation issues of EDB, a short description of EDB's environment, the ASF + SDF system will be presented.

In Chapter 2, the formalism ASF + SDF will be presented. Chapter 3 sketches the ASF + SDF system, an implementation of this formalism. Then, in Chapter 4, term rewriting in the ASF + SDF system is described in detail. A full report of the functionality of EDB, and of its user-interface can be found in Chapter 5. A description of the interface between EDB and the rest of the system is also given in Chapter 5. In Chapter 6, the techniques used in the Equation Debugger are compared with debugging techniques for programming languages. Finally, some concluding remarks are made in Chapter 7.

The ASF + SDF system is an integrated software system, with as purpose the automatic generation of interactive programming environments. A full description of this system can be found in [Kli90]. Programming environments are generated from modular formal language specifications, and changes in these specifications are processed incrementally. Each module of such a specification consists of two parts. In the first part, the syntax of a language is described using the syntax definition formalism SDF [HHKR89]. In the second part of a module, the semantics of the language under consideration are defined in the Algebraic Specification Formalism, ASF [BHK89].

In an SDF-definition, both the concrete and abstract syntax of a language are specified. The concrete syntax is expressed by regular and context-free grammars. In SDF a fixed mapping is defined from non-terminals (sorts) and grammar rules in the concrete syntax, to sorts and functions in the abstract syntax. Using this mapping, sentences (text strings) in the concrete syntax, can be mapped on their corresponding terms (abstract syntax trees) in the abstract syntax. In this way, SDF provides its users with a correspondence between the concrete and abstract syntax of a language.

If a sentence corresponds to more than one abstract syntax tree, i.e., the grammar is ambiguous, the ambiguities can often be solved by declaring priorities in the SDF-definition. These can, for example, be used to handle the precedence of arithmetic operators in a language. Furthermore, functions can be declared as left-, right- or non-associative. Language definitions in SDF are modular, generally, a definition consists of a set of modules. A module can use the syntax of another module, by importing it. Variables can be used as placeholders for subterms. Sorts, lexical and context-free grammar rules can be declared as hidden, in which case, the syntax associated can only be used in the module that defines them, it can not be used by importing modules.

As mentioned before, the ASF part of an ASF + SDF specification defines the semantics of a language. This is accomplished by using a set of conditional equations. These equations define equalities on terms of the same sort, and are viewed as rewrite rules transforming the left-hand-side of an equation into the corresponding right-hand-side. A list of conditions may be associated with each equation. Each condition consists of an equality or inequality on terms of the same sort. An equation is said to *match* with a term, when for all variables in the left-hand-side of an equation, a binding can be found, such that instantiating the variables with those bindings yields the given term. When an equation matches with a term, the conditions of that equation are checked. If they all succeed, possibly by binding more variables, the equation is applicable. Now, the variables in the right-hand-side of the equation may be instantiated using the bindings established so far, and the result may be substituted for the original term.

In the ASF + SDF system, equations are implemented as term rewriting rules, that is, they are only interpreted from left to right. The (sub)term, for which some equation is applicable, is called the *redex*. The application of an equation is called *rewriting*. If the redex is a subterm of a larger term, its context does not change. The process of rewriting continues until no more applicable rewrite-rules can be found. The term is then in *normal form*. The algorithm that is used to select a redex in a given term is called the *reduction strategy*. In the current implementation of the ASF + SDF system, a leftmost-innermost reduction strategy is used for the selection of redexes. Topics such as the reduction strategy, and the evaluation of conditions will be addressed in detail in Chapter 4.

The ASF + SDF system is a system that allows the user to create a specification, written in the combined formalism ASF + SDF. From this specification, a programming environment is generated, consisting of a syntax-directed editor and other tools, such as typecheckers and program-evaluators. In the system, a number of components can be distinguished, each with a definite task. The most important of these components are listed below.

- The Syntax Manager (SM) is responsible for the syntax part of specifications.
- The Equation Manager (EQM) is the component that handles the rewriting process of terms.
- The Module Manager (MM) controls the modular structures, defined in the specification.
- Various instances of the Generic Syntax-directed Editor (GSE) enable the user to edit terms and specifications in a manner, that combines both text- and structure editing. Instances of GSE are always parameterized with a language definition in SDF. GSE-instances are currently used in the ASF + SDF system for:
    (1) Editing terms in the language for which the user has supplied a specification. These GSE-instances are called term-editors, and are parameterized with the SDF-part of the user's specification.
    (2) Editing the syntax-part of specifications. These GSE-instances are parameterized with an SDF-specification of SDF itself.
    (3) Editing the equations part of specifications. Here, the GSE-instance is parameterized with a language definition that was derived from the user's definition, by extending it with the basic syntax for conditional

equations. In the ASF + SDF system the combination of (2) and (3) is called a module-editor.

(4) Finally, other instances of GSE will be used during the debugging-process, for entering breakpoints, and for pretty-printing partially rewritten terms. These GSE-instances will be discussed in Chapter 5.

A general overview of GSE's main features can be found in Section 3.3. In [Koo91], GSE is described in detail.

The Equation Debugger is a new component of the ASF + SDF system, to be used for debugging the semantics part of specifications. The main objectives of EDB are:

- Providing the user with a powerful means to detect errors in the equations part of a specification, in a highly interactive manner.
- Tracing, i.e., visualizing the rewriting process. A solution to this will be given in the form of pretty-printing partially rewritten terms, in which the current redex is indicated.

A key issue in the implementation of EDB is that debugging is done interactively, during the rewriting process. A basic design decision is that the user can only *observe* the rewriting process, and is not allowed to *change* terms during rewriting.

2. The formalism ASF + SDF

2.1 Introduction

In this chapter, the formalism ASF + SDF is presented by means of a sample specification. This specification contains a set of modules, defining the syntax of a small language of arithmetic expressions. In addition, the static and dynamic semantics of this language will be specified (corresponding to typechecking and evaluation, respectively). Usually, larger ASF + SDF specifications conform to the following modular structure :

- A set of basic modules defines the format of layout, and the datatypes used in the specification.
- A module for the syntax of the language.
- A typecheck module, expressing the static semantics of the language.
- Lastly, a module that specifies the dynamic semantics (evaluation).

In Section 2.2, the backgrounds of the formalisms ASF and SDF are summarized. Section 2.3 illustrates an informal explanation of our expression-language. This is followed by a description of the basic modules of the specification in Section 2.4. The module defining the syntax of our language is explained in Section 2.5. In Section 2.6, a typechecker for expressions is presented. Finally, in Section 2.7 we will describe the module, that defines an expression-evaluator. While presenting the modules of our specification, the features of ASF + SDF will be discussed in some detail. The example of expressions, as presented in this chapter, will be used throughout this report to illustrate the various features of EDB.

2.2 ASF + SDF

In the Syntax Definition Formalism SDF [HHKR89] a modular specification defines both the concrete syntax and the abstract syntax of a language. An important feature of SDF is the implicit connection between syntax and semantics. This is accomplished by a fixed mapping from parse-trees to abstract syntax trees (terms). Parse-trees are constructed by parsing text-strings according to the concrete syntax. The leaves of a parse-tree are the lexical tokens, that, when put in sequence, form the sentence that was parsed. The interior nodes of the parse-tree are the non-terminals of the grammar. Abstract syntax trees consist of the application of functions on constants, variables, and other functions. The relation between parse-trees, and their associated abstract syntax trees, will be illustrated in Section 2.4.

When combining SDF with the Algebraic Specification Formalism, ASF [BHK89], ASF provides algebraic semantics for SDF, through a set of conditional equations, over abstract syntax trees generated by SDF. Each module in the combined formalism ASF + SDF, contains both an SDF part, and an ASF part. The structure of such a module is outlined below, in Figure 1.

```
module M
  exports   -- syntax visible to importing modules.
    sorts S1 S2 S3
      -- declaration of sorts (non-terminals).
    lexical syntax
      -- rules defining the lexical syntax.
    context-free syntax
      -- definition of grammar-rules in the concrete
      -- syntax, and corresponding functions in the
      -- abstract syntax.
    variables
      -- declaration of naming schemes for variables
  imports M1 M2 M3    -- list of imported modules
  hiddens -- syntax visible only inside this module.
    sorts S4 S5 S6
      -- see exports section.
    lexical syntax
      -- see exports section.
    context-free syntax
      -- see exports section.
    variables
       -- see exports section.
  priorities
      -- definition of priority-relations between
      -- rules in the context-free syntax sections.
  equations
    -- conditional equations over abstract syntax.
```

Figure 1. The global structure of an ASF + SDF module.

2.3 An example: the specification of arithmetic expressions

In this example, expressions are constructed from integers constants and identifiers, and operators for addition, subtraction, and multiplication. In order to keep our example reasonably small, the common `succ - 0` representation is used for integers. Here, `0` represents the integer 0, `succ ... succ 0` (n times `succ`) represents the integer n, and `pred ... pred 0` (m times `pred`) represents -m. If constant values are used in expressions, the constructor-function `const` must be applied to their integer value. The precedence of functions is, in descending order:

- `( )`
- `let`
- `*`
- + and -

In a `let`-expression, identifiers *Id1,...,Idn* may be initialized for use in an expression `<EXP>`. Such a `let`-expression takes the form :

`let` *Id1=Value1,...,Idn=Valuen* `in <EXP>`.

Constraints, such as for example the fact that all identifiers should be declared, will be imposed by the expression-typechecker, in Section 2.6. `Let-constructs` can be nested, in which case the following scope-rule applies. The value of an identifier in an expression is the value assigned to it in the nearest surrounding `let`. For example, the correct value of the expression in Figure 2 is `succ succ succ succ 0`.

```
let a=0, b=succ 0
in
  let a=succ succ 0, c=succ succ succ 0
  in
    ( a * c - b * const(succ succ 0) )
```

Figure 2. A sample expression.

The typecheck-function takes an arbitrary expression as argument, and yields true if and only if :

(1) every identifier used in the expression is declared.

(2) no identifier occurs more than once in the same declaration-list.

If typechecking an expression succeeds, applying the evaluate-function to that expression returns its value. Otherwise, the expression itself is returned unaltered.

2.4 Basic Modules

Below, we will start with the basic module `Layout` as a first example. In module `Layout` (see Figure 3), the format of layout in our language of expressions is defined. Space-, tab- and newline-characters are layout. (The backslash symbol \ is an escape-mechanism for non-printable characters.) A general description of **`lexical syntax`** sections in ASF + SDF specifications will be given shortly.

```
module Layout
  exports
    lexical syntax
      [ \t\n]              ->    LAYOUT
```

Figure 3. An ASF+SDF specification of layout.

In module Booleans (see Figure 4), a sort BOOL is declared, and constants true and false of this sort are defined. There are two interpretations for sort-names declared in a **sorts** section. The first corresponds to non-terminals in the concrete grammar. The second interpretation is that of sorts in the algebra of terms (abstract syntax), associated with the specification. The sorts LAYOUT (as used in module Layout), and CHAR (not used in our example) are predefined, and do not have to be declared. On sort BOOL, two left-associative operators and and or, and a unary operator not are defined, representing the standard operations on booleans. The syntax associated with module Layout is imported into module Booleans, by listing it after the **imports** keyword. In the **priorities** section, the usual priorities are expressed: not has the highest priority, then and, and finally or. A bracket-function is defined to overrule these priorities, when necessary. Note that in equation BOOL5 the semantics of or are expressed using De Morgan's Law, in terms of not and and.

```
module Booleans
  exports
    sorts BOOL
    context-free syntax
      true                 ->    BOOL
      false                ->    BOOL
      BOOL and BOOL        ->    BOOL {left}
      BOOL or BOOL         ->    BOOL {left}
      not BOOL             ->    BOOL
      "(" BOOL ")"         ->    BOOL {bracket}
    imports Layout
    variables
      B[0-1]*              ->    BOOL
  priorities
     not > and > or
  equations
    [BOOL1] not true = false
    [BOOL2] not false = true
    [BOOL3] true and B = B
    [BOOL4] false and B = false
    [BOOL5] B0 or B1 = not ( not B0 and not B1 )
```

Figure 4. An ASF+SDF specification of Booleans.

As mentioned, we will describe the **lexical syntax** sections of an ASF + SDF specification. In these sections, lexical notions of the language specified are defined, using regular grammars. In the rules of the **context-free syntax** sections, these lexical notions will be utilised. The grammar of the **lexical**

syntax section may contain character classes, and repetition-operators * and +, denoting zero-or-more and one-or-more repetitions, respectively. An example can be found in module `Elements`, which is presented below (see Figure 5). Here, the elements of expressions are specified, i.e., integers and identifiers. In the rule

```
[a-z][a-z0-9_]*        ->    ID,
```

the lexical notion `ID` (identifier) is defined as a string starting with a letter, followed by zero or more letters, digits or underscore symbols.

As stated before in Section 2.2, a dual view exists on the syntax defined by a specification. In the context-free syntax section, both the concrete syntax, and the abstract syntax are defined.

- The concrete syntax corresponds to interpreting sorts as non-terminals, and reading the rules from right to left. Actually, a BNF-grammar is derived from the rules in the specification, when parsing of sentences is necessary. According to this grammar parse-trees are then constructed. Parse-trees show how a particular sentence can be derived from a grammar. The interior nodes of a parse-tree are the non-terminals of the grammar that is used, and the leaves of a parse-tree are the lexical tokens. When these tokens are put in sequence, they form the sentence that was parsed originally. In Figure 6a, an example of a parse-tree is presented.
- The abstract-syntax defined by a specification corresponds to the domain-interpretation of sorts. Here, a rule in the context-free syntax section is seen as the declaration of a function. The sort-names occurring in the left-hand-side of a context-free syntax rule correspond to the types of the arguments, and the sort-name that forms the right-hand-side defines the result-type of the function. Abstract syntax trees are constructed according to this abstract syntax. Abstract syntax trees only contain the essential information, necessary to describe a sentence, offering a much more concise representation of sentences than parse-trees. Abstract syntax trees consist of the application of functions to other functions, constants and variables. In the SDF reference manual [HHKR89], the algorithm that maps a parse-tree on its corresponding abstract syntax tree can be found. An example of an abstract syntax tree is shown in Figure 6b.

If we consider for example the rule `add INT INT -> INT`, as found in module `Elements`, its two interpretations are :

(1) from a non-terminal `INT`, we can derive the terminal `add` followed by two new `INT` non-terminals.

(2) the application of a function `add` on two operands of type `INT` yields a result of type `INT`.

In the **equations** section of an ASF + SDF module, semantics are associated with the syntax, by way of a set of conditional equations. First we will examine some simple unconditional equations. For each sort `S` in the specification, unconditional equations of the form `[Tag] S1 = S2` can be written, where `S1` and `S2` are terms according to the abstract syntax of sort `S`, and `Tag` is a name that is attached to the equation. Uniqueness of tags is not demanded, but is strongly recommended for debugging purposes, that we will describe in Chapter 5. As an example, we consider the equations with tags `Int3`, `Int4` and `Int5` of module Integers (Figure 5), in which the semantics of integer-addition are defined. Using standard arithmetic notation we see that :

- equation `Int3` corresponds to : $x + 0 = x$

- equation `Int4` corresponds to : $x + (y+1) = (x+1) + y$
- equation `Int5` corresponds to : $x + (y-1) = (x-1) + y$

A list of conditions (separated by commas) may be associated with an equation, each condition being an equality or an inequality of terms of the same sort. So far, we have not yet used an equation with conditions, but an example is equation `Tc9` from module `Exp-tc` (Figure 8), to be presented in Section 2.6.

```
[Tc9]  Id0 != Id1
       ========================================
       [ Id0 ] in (Id1 Ids) = [ Id0 ] in (Ids)
```

In **`variables`** sections, naming schemes for variables are defined. Declarations in this section define the lexical structure of names of variables, using regular grammars. For example, in module `Elements` (see Figure 5), the names of variables of sort `INT` are defined to consist of the letters `Int`, followed by zero or more digits `0` or `1` :

```
Int[0-1]*          ->    INT
```

```
module Elements
  exports
    sorts INT ID
    lexical syntax
      [a-z][a-z0-9_]*       ->    ID
    context-free  syntax
      "0"                   ->    INT
      succ INT              ->    INT
      pred INT              ->    INT
      add INT INT           ->    INT
      sub INT INT           ->    INT
      mul INT INT           ->    INT
    variables
      Int[0-1]*             ->    INT
  imports Layout
  equations
    [Int1]  succ pred Int = Int
    [Int2]  pred succ Int = Int
    [Int3]  add Int 0 = Int
    [Int4]  add Int0 succ Int1 = add succ Int0 Int1
    [Int5]  add Int0 pred Int1 = add pred Int0 Int1
    [Int6]  sub Int 0 = Int
    [Int7]  sub Int0 succ Int1 = sub pred Int0 Int1
    [Int8]  sub Int0 pred Int1 = sub succ Int0 Int1
    [Int9]  mul Int 0 = 0
    [Int10] mul Int0 succ Int1 = add mul Int0 Int1 Int0
    [Int11] mul Int0 pred Int1 = sub mul Int0 Int1 Int0
```

Figure 5. The basic elements of expressions.

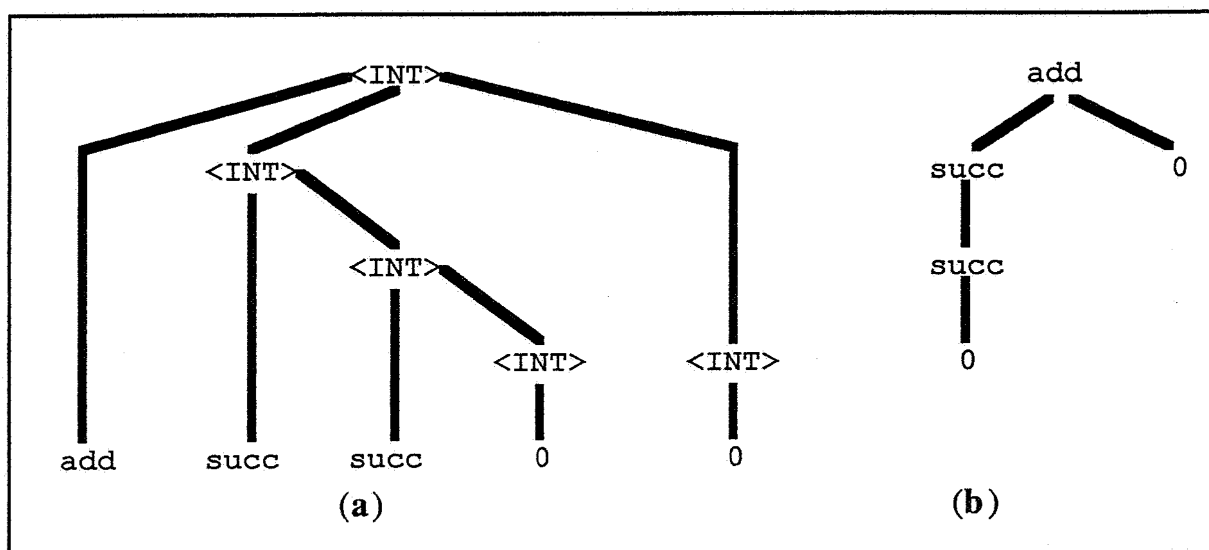


Figure 6. Parse-tree (a) and corresponding abstract syntax tree (b) for the sentence 'add succ succ 0 0'.

2.5 The syntax of expressions

Having specified the set of basic modules, we are able to present the module defining the syntax of our expression-language. This module, `Expressions`, can be found below, in Figure 7. Two sorts, `DECL` and `EXP` are introduced in this module. The sort `DECL` is related to one declaration in a `let`-construct. The other sort, `EXP`, corresponds to the actual expressions.
We will now give a brief description of the grammar rules of module `Expressions`. In the grammar rule `ID = INT -> DECL`, is expressed, that each declaration consists of an identifier, followed by =, and an integer. The grammar rule `const "(" INT ")" -> EXP` shows that a constructor-function `const` allows us to use integer-constants in expressions. No such constructor function is necessary for using identifiers in expressions, as stated by the rule `ID -> EXP`. This kind of grammar rule, that in fact defines a subsort, is called a *chain-function* or *injection*.
In the grammar rule `EXP "+" EXP -> EXP {left}`, an associativity-attribute is used. This defines the + operator, working on expressions, to be left-associative. Likewise, right- and non-associativity of binary functions `S op S -> S` can be expressed with the attributes `{right}`, and `{non-assoc}`.
Another attribute, `{bracket}`, is used to define *bracket-functions*, which are needed to overrule priority- and associativity-constraints. An example of this is the rule `"(" EXP ")" -> EXP {bracket}`. Bracket functions are used for grouping purposes only, they do not contribute to the abstract syntax.
In the rule `let { DECL "," }+ in EXP -> EXP`, the *list-sort* `{ DECL "," }+` is used. Generally, for every sort `S` defined in a specification, the list-sorts `S*`, and `S+` are defined implicitly, representing zero-or-more , and one-or-more repetitions of sort `S`, respectively. Furthermore, for lists with separators (in our example a comma), the list-sorts `{ S sep }*`, and `{ S sep }+` can be used. In the concrete syntax, these sorts represent lists of zero-or-more, and one-or-more repetitions of sort `S`, separated by the separator `sep`. In the abstract syntax, no separators are placed between the list-elements in both cases. Here, the list-elements are children of a node, that indicates both the sort of the elements of the list, and the separator

used. List-sorts are not allowed on the right-hand-side of context-free syntax rules. However, this imposes no restrictions, since list-sorts on the right-hand-side can always be simulated using an auxiliary sort.
In the **priorities** section, priority-relations between sets of context-free syntax rules are defined, in priority-chains. These priority-chains consist of a list of (sets of) context-free syntax rules (without their attributes), separated by either < or >. Rules in priority-chains may be abbreviated, by writing only the terminals occurring in the left-hand-side. However, this is only allowed, if the abbreviations thus found are unique.
If we consider the priority-chain in module `Expressions` (see Figure 7), we see that `let-constructs` have the highest priority, followed by `*`, and the set of + and -.
Associativity-attributes can also be attached to groups of operators, by placing the appropriate attribute in front of a set of operators in a priority-chain. An example of this is the declaration of + and - as a left-associative group, in the last line of the **priorities** section of module `Expressions`.
Group-associativity declarations define how operators with the same priority associate with each other. If we consider, for example, the string `a + b - c`, then the declaration of + and - as a left-associative group, only leaves the interpretation `(a + b) - c`.

```
module Expressions
  exports
    sorts DECL EXP
    context-free  syntax
      ID "=" INT                    ->    DECL
      const "(" INT ")"             ->    EXP
      ID                            ->    EXP
      EXP "+" EXP                   ->    EXP {left}
      EXP "-" EXP                   ->    EXP {non-assoc}
      EXP "*" EXP                   ->    EXP {left}
      "(" EXP ")"                   ->    EXP {bracket}
      let { DECL "," }+ in EXP      ->    EXP
  imports Elements Booleans
  priorities
    let in  >
    "*"   >
    { left: "+", "-"  }
```

Figure 7. Syntax of expressions.

2.6 A typechecker for expressions

Now, we will present the module `Exp-tc` (see Figure 8), which defines the static semantics of expressions. The type-constraints that will be verified are :

(1) Expressions do not contain undeclared identifiers.
(2) The declaration-list of one `let`-construct does not contain the same identifier more than once.

We will use a scheme, that is often used for specifying typecheckers. Declarations are entered in a so-called *type-environment*, that is passed as an argument to the actual typecheck function. When a statement is typechecked, the type-constraints

are verified, by comparing the type-information of the statement, with the corresponding information in the type-environment. In our example, type-environments only contain the identifiers, that are declared in `let`-constructs.

```
module Exp-tc
  exports
    sorts TENV
    context-free  syntax
      tc "[" EXP "]"                               ->    BOOL
      "(" ID* ")"                                  ->    TENV
      "[" EXP "]" in TENV                          ->    BOOL
      add-to-tenv "(" { DECL "," }* "," TENV ")"
                                                   ->    TENV
      doubles "(" TENV ")"                         ->    BOOL
    variables
      E[1-2]*                                 ->    EXP
      Int                                     ->    INT
      Tenv                                    ->    TENV
      Id[0-1]*                                ->    ID
      Ids                                     ->    ID*
      Decls                                   ->    { DECL "," }*
  imports Expressions
  equations
    [Tc1]  tc[ E ] = [ E ] in ()
    [Tc2]  doubles( add-to-tenv( Decls , () ) )   = true
           ============================================
           [ let Decls in E ] in Tenv = false
    [Tc3]  doubles( add-to-tenv( Decls , () ) ) != true
           ============================================
           [ let Decls in E ] in Tenv =
               [ E ] in add-to-tenv( Decls , Tenv )
    [Tc4]  [ E1 + E2 ] in Tenv =
             [ E1 ] in Tenv and [ E2 ] in Tenv
    [Tc5]  [ E1 - E2 ] in Tenv =
             [ E1 ] in Tenv and [ E2 ] in Tenv
    [Tc6]  [ E1 * E2 ] in Tenv =
             [ E1 ] in Tenv and [ E2 ] in Tenv
    [Tc7]  [ Int ] in Tenv = true
    [Tc8]  [ Id ] in (Id Ids) = true
    [Tc9]  Id0 != Id1
           ====================================
           [ Id0 ] in (Id1 Ids) = [ Id0 ] in (Ids)
    [Tc10] [ Id ] in () = false
    [Tc11] add-to-tenv( , Tenv) = Tenv
    [Tc12] add-to-tenv( Id=Int,Decls , (Ids) ) =
           add-to-tenv( Decls , (Id Ids) )
    [Tc13] Tenv = (Ids0 Id Ids1 Id Ids2)
           ============================
           doubles( Tenv ) = true
```

Figure 8. An expression-typechecker.

A sort `TENV` is declared, indicating type-environments. Type-environments are lists of identifiers between brackets. The purpose of the function `add-to-tenv` is adding elements of declaration-lists to a type-environment, returning a new type-

environment. The function doubles checks, whether a type-environment contains the same identifier more than once. It returns a boolean value. The function tc "[" EXP "]" -> BOOL, is the actual typecheck-function. It uses the auxiliary typecheck-function "[" EXP "]" in TENV -> BOOL, that typechecks an expression in a particular type-environment.
In equation Tc1, typechecking is defined to start in an empty type-environment. In equations Tc2 and Tc3, constraint (2) is checked. If a variable occurs twice, typechecking fails, and false is returned. Otherwise, all variables are added to the type-environment. The actual verification of the uniqueness of identifiers in a TENV is done in rule Tc13, where list-matching is used to check if some element occurs twice. In equations Tc11 and Tc12 the semantics of add-to-tenv are expressed, in a recursive way. In equations Tc4 - Tc6, composite expressions are checked, and in equations Tc7 - Tc9 basic expressions are handled (integer constants and identifiers).

2.7 An evaluator for expressions

This chapter will be completed by a brief description of an expression-evaluator (see Figure 9). Given a typechecked expression, the evaluate-function will determine the (integer) value of that expression.
We will use a similar scheme for evaluation, as the scheme we presented for type-checking, in the previous section. This time, declarations are put in a *value-environment*, that is passed as an argument to the evaluate-function. When an identifier is evaluated, its value is searched for in the value-environment.
The sort VENV, as introduced in this specification denotes a value-environment, which consists of an ordered list of identifier-value pairs (sort PAIR), between brackets. In module Exp-ev, a function add-to-venv is introduced for adding elements of declaration-lists to a value-environment. The actual evaluate-function is here ev "[" EXP "]" -> INT. This function uses the auxiliary function "[" EXP "]" in VENV -> INT for evaluating expressions in a particular value-environment. Observe, that a strong similarity exists between this evaluate-function, and the typecheck-function of the previous paragraph.
Conditional equation Ev1 defines that an expression should only be evaluated, if both typechecking constraints are met. In equation Ev2, identifiers of a let-declaration are added to the environment, and the expression contained in the let-construct, is evaluated in this environment. In order to implement the scope-rules, as discussed before, the following strategy is adopted : declaration-lists of let-constructs are always added to the left side, in the value-environment. When a value of an identifier is needed, the list is scanned from left to right. As a result, the first value we find for an identifier, is the correct one. This scanning of value-environments is expressed in equations Ev4 and Ev5. Equation Ev3 says, that the value of an integer-subexpression does not depend on its value-environment. Finally, equations Ev6 - Ev8 map the additions, subtractions and multiplications of expressions on their corresponding integer-functions.

```
module Exp-ev
  exports
    sorts PAIR VENV
    context-free syntax
      ev "[" EXP "]"                              ->   INT
      "(" PAIR* ")"                               ->   VENV
      add-to-venv "(" { DECL "," }* "," VENV ")"
                                                  ->   VENV
      ID ":" INT                                  ->   PAIR
      "[" EXP "]" in VENV                         ->   INT
    variables
      E[1-2]*                            ->   EXP
      Int[0-9]*                          ->   INT
      Id[0-9]*                           ->   ID
      Decls[0-9]*                        ->   { DECL "," }*
      Pairs[0-9]*                        ->   PAIR*
      Venv[0-9]*                         ->   VENV
  imports Exp-tc
  equations
    [Ev1]   tc[ E ] = true
            =====================
            ev[ E ] = [ E ] in ()
    [Ev2]   [ let Decls in E ] in Venv =
              [ E ] in add-to-venv( Decls , Venv )
    [Ev3]   [ Int ] in Venv = Int
    [Ev4]   [ Id ] in (Id:Int Pairs) = Int
    [Ev5]   Id0 != Id1
            ===========================================
            [Id0] in (Id1:Int Pairs) = [Id0] in (Pairs)
    [Ev6]   [ E1 + E2 ] in Venv =
              add [ E1 ] in Venv [ E2 ] in Venv
    [Ev7]   [ E1 - E2 ] in Venv =
              sub [ E1 ] in Venv [ E2 ] in Venv
    [Ev8]   [ E1 * E2 ] in Venv =
              mul [ E1 ] in Venv [ E2 ] in Venv
    [Ev9]   add-to-venv( , Venv ) = Venv
    [Ev10]  add-to-venv( Id=Int,Decls , (Pairs) ) =
            add-to-venv( Decls , (Id:Int Pairs) )
```

Figure 9. An evaluator for expressions.

3. The ASF + SDF system

3.1 Overview

In this chapter, we will describe the ASF + SDF system. This is a system that implements the formalism ASF + SDF, as described in the previous chapter. The ASF + SDF system is being developed in connection with the ESPRIT projects GIPE (Generation of Interactive Programming Environments), and GIPE II. The ASF + SDF system is based on CENTAUR [Centaur], a generic set of tools for building environment generators. From this set of tools, the Virtual Tree Processor (VTP) is used for manipulating abstract syntax trees, and abstract syntax trees are pretty-

printed using a system for pretty-printing nested objects, called Figue. The implementation language of the ASF + SDF system is LeLisp [LeLisp87].
In order to develop specifications, the user is provided with the following functionality :

- Creation of a new specification.
- Addition of a module to a specification.
- Modification/deletion of a module.
- Reading/saving a specification from/to file(s).
- Testing the semantics of a specification.

The most significant features of the ASF + SDF system are :

- Fully incremental processing of changes made in a specification.
- The use of the same syntax-directed editor for editing terms and modules, thus providing the user with a uniform interface.

3.2 The ASF + SDF meta-system

Figure 10 shows the window of the ASF + SDF meta-system, which is the top-level of the user-interface of the ASF + SDF system. This window has four menus attached to it. The window itself is used for displaying error-messages. The functionality provided by the menus is described below.

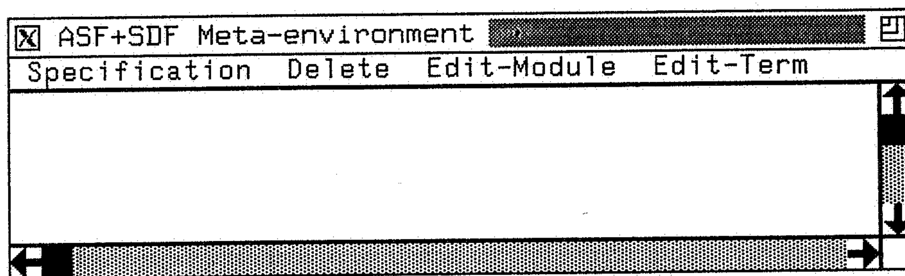


Figure 10. The window of the ASF + SDF meta-system.

The `Specification` menu contains the items `add`, `clear`, `save` and `quit`. `Add` allows the user to create a new module, or read an existing module with its imported modules from file. The other three items enable the user to re-initialize the system, save all modules, and leave the system, respectively.
The other three menus, `Delete`, `Edit-Module`, and `Edit-Term` always contain the same items, they contain one item for every module in the current specification. The name of each item is the name the corresponding module, and the items are ordered alphabetically in their menus.
Selection of an item of the `Delete` menu deletes the corresponding module. If an item of the `Edit-Module` menu is selected, a *module-editor* for the corresponding module is started. In Section 3.5, the subject of module-editors will be illustrated with an example. The `Edit-Term` menu provides items, that will start *term-editors* for the associated modules. A term-editor enables the user to edit sentences over the concrete syntax of the selected module. Furthermore, the semantics of the module can be tested by rewriting the term, that is derived from the edited sentence. In Section 3.4, term-editors will be described, and in Chapter 4 the rewriting of terms will be studied in detail.

3.3 The Generic Syntax-directed Editor GSE

The generic syntax-directed editor GSE combines text- and structure-editing. Various *instances* of GSE are used in the ASF + SDF system. Every instance of GSE is parameterized with an SDF language-definition. This SDF-definition is used for constructing abstract syntax trees, which are needed for structure editing purposes. The following instances of GSE are used in the ASF + SDF system:

- Term-editors are GSE-instances, used for editing terms over the syntax of a particular module. In Section 3.4, term-editors are described.
- Module-editors actually consist of two GSE-instances. The first is used for editing the syntax part of a module. The second instance enables the user to edit the equations part of a module. In Section 3.5, module-editors are described.

Before the main features of GSE are discussed, we will give a short description of structure-editing. Structure-editing is always performed on the abstract-syntax tree representation of the text, that is edited. Two important concepts of structure-editing are:

- Placeholders for subterms (holes).
- The focus. This is a part of the text, that is directly related to a subtree of the abstract syntax tree. The focus indicates the subtree, that is currently under consideration.

A structure-editing action can replace a placeholder of sort `S`, on which the focus is positioned, by a syntactically correct term of sort `S`. If the focus is positioned on a list-element, a structure-editing action can insert a new placeholder for a list-element immediately before or after the focus. Structural movements enable the user to move the focus, by navigating in the tree. Structure-editing in GSE-instances will be addressed in more detail shortly.

At this point, we will consider how text- and structure-editing are combined in GSE. Text-editing in a GSE-instance can only take place inside the focus. Moving the focus is accomplished by either clicking the mouse somewhere in the text, or by structural movements, such as go-up-in-tree, and go-to-next-child. When such attempts of moving the focus are made, the text inside it is always parsed. If parsing succeeds, the focus is moved as intended. Otherwise, an error-message is presented to the user in a separate window, and the new focus is determined as follows:

- If the focus-move was initiated by a mouse-click at some character in the text, the new focus will be positioned on the smallest subtree, that includes both the old focus, as well as the character pointed at.
- When a structural move go-up-in-tree is attempted, and a parsing-error occurs, the focus is moved as indicated.
- Otherwise (a parsing-error occurred, and one of the other structural movements was attempted), the focus is left unchanged.

As a result of the strategy above, the text outside the focus is always syntactically correct.

Next, we will return to the subject of structure-editing in GSE. In the ASF + SDF system, placeholders are called *meta-variables*. Meta-variables are represented as

strings of the form <S>, where S is a sort in the specification parameterizing the GSE-instance. A meta-variable <S> may be included anywhere in the text, where a parse of sort S is expected. If the focus is positioned on a meta-variable, the following actions are possible :

- Text is typed. In this case, the meta-variable will disappear, to be replaced by the text just entered.
- Items in one of the menus of GSE allow one to replace the meta-variable, according to any of the grammar rules of the proper sort.

Having described the manner in which GSE combines text- and structure-editing, we now turn to the user-interface of GSE. This consists of the following parts:

- A scrollable window, containing the text. The focus is indicated by lines surrounding it. A cursor indicates the place, where new text will appear.
- A menubar, containing the options □, `tree`, `text`, `expand`, and `help`.
- An optional column of buttons, on the left side of the window.

As an example of a GSE-instance, in Figure 11 we present a term-editor showing a term in module `Expressions` (Chapter 2).

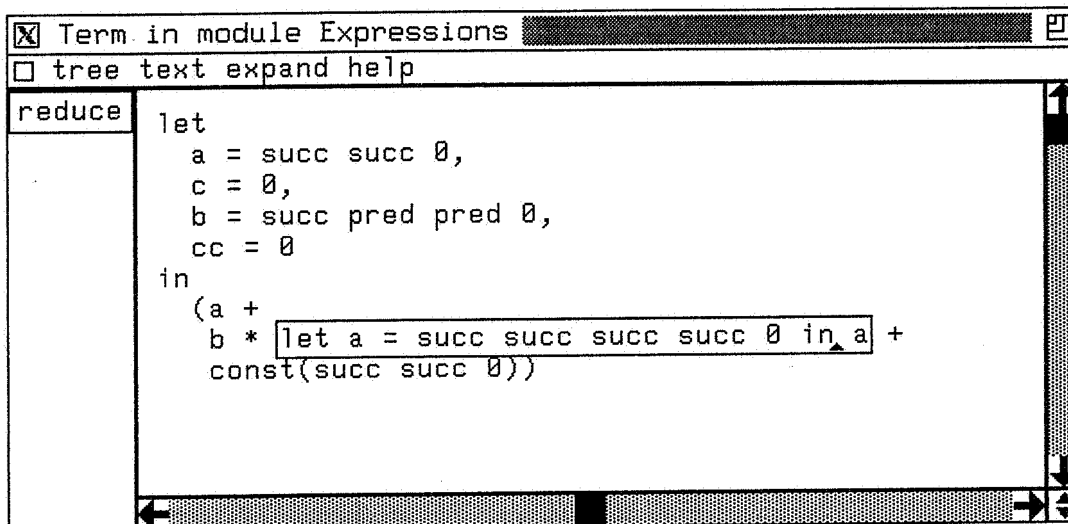


Figure 11. Example of a GSE-instance : a term-editor.

Below, we will summarize the functionality, that is provided by the menubar of a GSE-instance.

- When the □ button is pressed, the editor is left. If the text has been changed, the user is asked whether the text should be saved to file.
- The `tree` menu provides items for structural movements in the abstract syntax tree, and for insertion of meta-variables in lists. Before performing these operations, the focus will be parsed. If parsing fails, only the `zoom out` action can be performed. In that case, all other actions leave the focus unchanged.

  The items `zoom in`, `zoom out`, `next child`, and `previous child` move the focus to the first child, to the parent, to the next child (the next subtree at the same level as the focus), and to the previous child, respectively. If such a focus move is not possible, the focus will remain unaltered.

If the focus is a list-element, the items `insert hole after` and `insert hole before` will insert a meta-variable in the list after the focus, and before the focus, respectively. If necessary, a separator will also be inserted. If the focus itself is not a list-element, but an ancestor of the focus is, the same actions as mentioned above will be performed, for the first ancestor of the focus that is a list-element.

- In the `text` menu, two items, `cut` and `paste`, for textual manipulations can be found.
  If a piece of text is selected, by dragging the mouse, `cut` removes it. Otherwise, the text of the focus is removed. The `paste` item pastes the most recent selection in the GSE-window, or in another X-window after the current position of the cursor.
- When the focus is positioned on a meta-variable, the `expand-menu` contains an item for every possible *expansion*. These expansions are the context-free syntax rules, and lexical output sorts, that may be substituted for that variable. Choosing an expansion, which is a context-free function, will result in replacing the meta-variable by the sequence of terminals and non-terminals on the left-hand-side of that context-free function. The other possibility is, that a choice is made for an expansion, which is the output-sort of a lexical function. Then, the meta-variable will disappear, and the user may enter a lexeme by typing it.
  If the focus is not positioned on a meta-variable, the `expand-menu` will only contain the message: `Non expandable sort : S`, with `S` the sort of the focus' subterm.
- The `help` menu contains the items `undo`, `cursor to error`, and `show cursor`. The first of these items tries to undo the last action of the user. `cursor to error` moves the cursor to the place, where a parsing-error occurred. Finally, `show cursor` scrolls the window until the cursor is visible in the middle of the screen.

In case of ambiguities that occur when a string is parsed, a separate disambiguator-window will pop up. This window will contain one item for every possible parse of the text. Another item shows the area, where the ambiguity was found. The user is asked to choose one of the alternatives. When the choice is made, the disambiguator-window will disappear, and parsing continues.

3.4 Term-Editors

As mentioned earlier, a term-editor is started by selecting an item of the `Edit-Term` menu of the meta-system. Term-editors are instances of GSE, parameterized with the syntax, defined by the selected module. By default, a term-editor contains one button, labeled `reduce`. Pressing this button results in rewriting the abstract syntax tree corresponding to the text in the editor, to its normal form. When the normal form is reached, it will be displayed in the window (console), from where the command was given to start the ASF + SDF system.

Other buttons can be added to a term-editor, by defining their semantics in the configuration-file of the ASF + SDF system. Two modules are associated with each button. The first module defines the term-editor, in which the button will appear.

The syntax and semantics of the second module are used when the button is pressed. We will not explain the use of this configuration file in this report, see however [Hen91].
In Figure 12 below, a term-editor over module `Expressions` is displayed. Two buttons, `tc` and `ev` are added. When the `tc` button is pressed, the current term in the editor, `E`, is passed to the typecheck function. The resulting term, `tc[E]` is rewritten, using the syntax and semantics of module `Exp-tc`. Likewise, pressing the `ev` button evaluates the current term, using module `Exp-ev`.

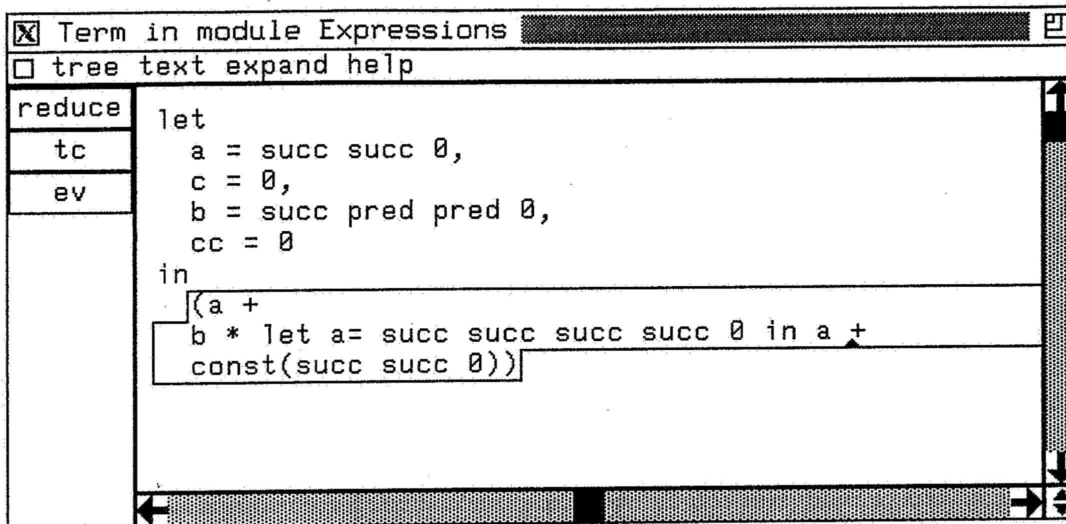


Figure 12. Term-editor with buttons `tc` and `ev`.

3.5 Module-Editors

A module-editor consists of the combination of two GSE-instances. The first instance, which is an editor for the syntax part of a module, is parameterized by an SDF-definition of SDF itself. The second GSE-instance is used for editing the equations part of a module. Here, the instance is parameterized with a language definition, that was derived from the users definition by extending it with the basic syntax for conditional equations. Below, Figure 13 shows a module-editor for module `Exp-ev`, that was presented in Section 2.7.

```
[X] Module Exp-ev
[] tree text expand help

%%
%%      Exp-ev
%%

   imports Exp-tc

   exports
      sorts PAIR VENV
      context-free syntax
         ev "[" EXP "]"                                  ->      INT
         "(" PAIR* ")"                                   ->      VENV
         add-to-venv "(" { DECL "," }* "," VENV ")"      ->      VENV
         ID ":" INT                                      ->      PAIR
         "[" EXP "]" in VENV                             ->      INT

equations
  [Ev1]   tc[ E ] = true
          =====================
          ev[ E ] = [ E ] in ()
  [Ev2]   [ let Decls in E ] in Venv = [ E ] in add-to-venv( Decls
  [Ev3]   [ const(Int) ] in Venv = Int
  [Ev4]   [ Id ] in (Id : Int Pairs) = Int
  [Ev5]   Id0 != Id1
          ===============================================
          [ Id0 ] in (Id1:Int Pairs) = [ Id0 ] in (Pairs)
  [Ev6]   [ E1 + E2 ] in Venv = add [ E1 ] in Venv [E2] in Venv
  [Ev7]   [ E1 - E2 ] in Venv = sub [ E1 ] in Venv [E2] in Venv
```

Figure 13. Module-editor for module Exp-ev.

3.6 The internal structure of the ASF + SDF system

Modularity is a key issue of the ASF + SDF system. In the ASF + SDF system, modularity is implemented through a selection mechanism. Components, like parsers, scanners, and term-rewriters are generated in a lazy and incremental way for the entire specification. According to the selection mechanism, first a module is indicated to be the current module. Then, as a result, only the parts of the component involved, corresponding to the current module, are selected for use. A full description of this selection-mechanism can be found in [Hen91].
The ASF + SDF system contains the following major components:

- The Syntax Manager (SM)
  The Syntax Manager maintains all information, regarding the syntax of specifications. It uses MPG, a lazy, incremental and modular parser generator, to generate parsers capable of handling arbitrary context-free grammars (based on Tomita's algorithm). Another component of SM is MSG, the lazy, incremental, and modular scanner generator, which can manage ambiguous regular expressions.

The syntax manager provides operations for the addition or deletion of a sort declaration, lexical function definition, context-free function definition, variable declaration, priority declaration, import, or equation. Furthermore, a module can be selected as the current module. This results in selecting those parts of the scanner and parser, that accept the grammar defined in the indicated module (including the visible grammar of imported modules). Finally, the grammar of the current module can be used to parse a given string.

- The Equation Manager (EQM)
  The Equation Manager is the component of the ASF + SDF system that handles the semantic aspects of specifications. It interprets equations as rewrite-rules. Details are discussed in Chapter 4. As in the case of SM, EQM is an incremental, and modular tool, using a global set of equations, and a selection mechanism.
  The functionality provided by EQM consists of the addition or deletion of an equation, given a particular module. Again, a module can be selected as the current module. As a result, all equations, which can be used in the context of that module, will be selected. Lastly, the set of selected equations can be used to rewrite a given term.

- The Module Manager (MM)
  The Module Manager maintains the import structure of a specification. To this end, it offers operations for the addition and deletion of modules. Another task of MM, is to provide an interface to the components SM and EQM. Therefore, it also offers operations for adding and deleting syntactic elements, for parsing a string, and for rewriting of a term. The latter two operations are propagated, to the appropriate components, SM and EQM, respectively.
  MM provides operations for the addition or deletion of a module to/from a specification. Furthermore, a sort declaration, lexical function definition, context-free function definition, priority declaration, variable declaration, import, or an equation can be added or deleted to/from a module. A module can be selected as the current module. Finally, the grammar rules and the equations of the current module can be used to parse a given string, and to rewrite a given term, respectively.

- The Generic Syntax-directed Editor (GSE)
  GSE has been described in the previous chapters. Changes made in the text of a module-editor are propagated to MM and SM. Pressing a `reduce` button in a term-editor will result in offering the created term to MM, and then to EQM.
  GSE provides operations for the creation of a new GSE-instance, given a language definition in SDF. Edit-operations can be performed, both textual and structural. Changes in the text are propagated to the appropriate components. Finally, the execution of functions, that are associated with buttons, is supported.

- The Equation Debugger (EDB)
  The Equation Debugger is the subject of this report. It is a new component of the ASF + SDF system, with the following tasks:
  (1) Providing the user with a means to debug specifications.
  (2) Tracing, i.e., visualizing the rewriting process.
  To this end, information regarding the rewriting of a term, has to be exchanged with EQM. Furthermore, some instances of GSE will be used for purposes of pretty-printing (unparsing), and editing breakpoints. This topic, the interface of EDB with the other components in the system, will be described in Chapter 5. In this chapter, the user-interface and functionality of EDB will be presented as well.

In Figure 14, the major components of the ASF + SDF system are shown. Arrows indicate paths of communication.

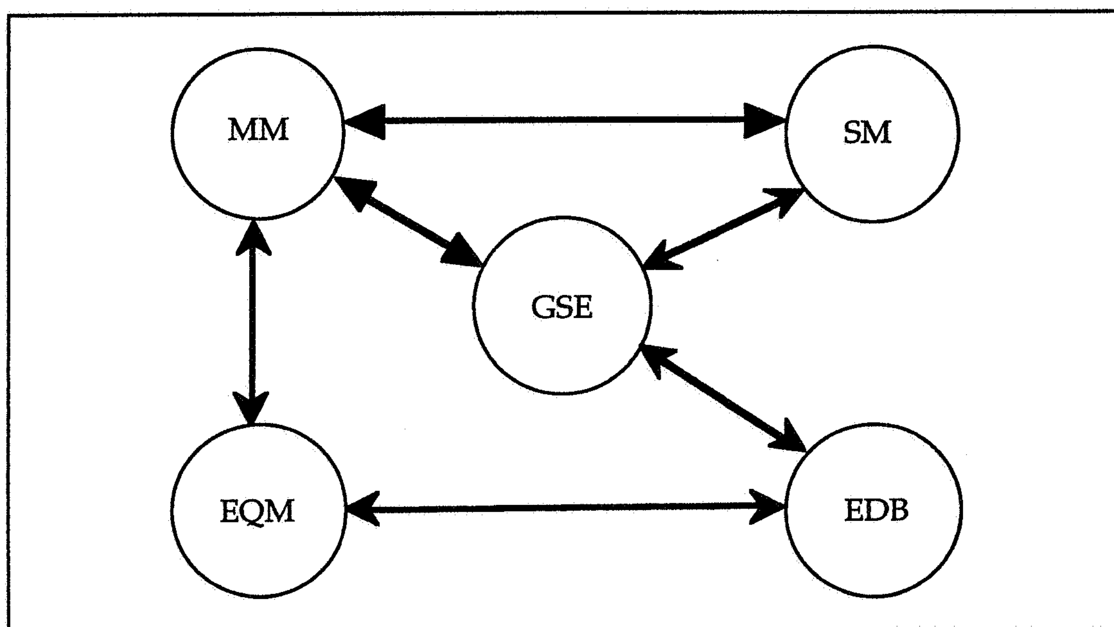


Figure 14. Global architecture of the ASF + SDF system.

4. Term rewriting in the ASF + SDF system

As mentioned earlier, EQM rewrites terms by interpreting equations as rewrite-rules. The interpretation of equations as a term rewriting system is obtained, by applying equations from left to right only. An equation and a term *match*, if a *binding*, i.e., a mapping from variables to terms exists, such that replacing the variables by their value results in the given term. After matching an equation with a term, the (possibly empty) list of conditions of that equation is checked.
A condition is an equality, or an inequality of terms of the same sort. Only if all conditions succeed, possibly by binding more variables, an equation is *applicable*. We will describe the testing of conditions in more detail shortly.
A rewrite-rule may only be applied to a term, if it is applicable, in that case all variables in the right-hand-side can be replaced by their corresponding value. We call the substitution of the bindings for the variables in the right-hand-side of an equation *instantiation*. The original term, for which some applicable equation was

found, is called the *redex*. We call the application of an equation a *rewrite-step*. A rewrite-step consists of replacing the redex associated with some applicable equation, by the *result* of applying that equation, i.e., the instantiated right-hand-side. If the redex is a subterm of a larger term, its context remains unaltered.

In the ASF + SDF system, the rewriting of a term is usually initiated by pressing the `reduce` button in a term-editor. (Other buttons in term-editors can also start the rewriting of a term. An example can be found in Section 3.4, where the `tc` and `ev` buttons start the rewriting of a term.)
Pressing the `reduce` button causes the module of the term-editor to be selected as EQM's current module. As a result, the rewrite-rules corresponding to that module will be selected from EQM's global set of rewrite-rules. Then, the term in the editor will be rewritten according to this subset of rewrite-rules. Below, the algorithm used by EQM for rewriting terms is described, in an informal way.

(a) A subterm is chosen for matching, in a leftmost-innermost fashion. This reduction-strategy will be commented upon shortly.

(b) Given the subterm, that was chosen in (a), the set of selected equations is searched. This search continues until
  (1) An equation is found with a left-hand-side that matches this subterm. In this case, we continue at (c).
  or
  (2) All equations have been searched, given this subterm. If more subterms can be tried, we continue at (a), with the next subterm. If all (sub)terms have been tried, and no match could be found, the term is in normal form, and rewriting ends.

  No assumptions should be made regarding the order, in which equations are tried for matching. A common misunderstanding is, that the equations are considered in the same order as in the specification. This is not necessarily the case.

(c) At this point, we have a match of a subterm and an equation, resulting in a set of bindings, for the variables occurring in the left-hand-side of that equation.
  If the equation has no conditions, the subterm is rewritten, by replacing it by the instantiated right-hand-side of the equation. After this rewrite-step we return to (a).
  If the equation does not have conditions, there are two cases:
  (1) All conditions succeed. (Possibly extending the set of bindings.) In this case, the term is rewritten, and we return to (a) to select the next subterm.
  (2) One of the conditions fails. Here, we return to (b), and continue the search for a match through the remaining equations, using the same subterm. Note: this situation is slightly more complicated if list-sorts are considered. We will now look at this in some detail.

In Chapter 2, list-sorts were described. Although list-sorts make the job of writing specifications much easier, they introduce some complications in the matching of terms with equations. To see this, we consider an equation with two consecutive list-variables, `X` and `Y`, of sort `S*` in its left-hand-side. When a match of `XY` with the

list `ab` is considered (`a` and `b` are both constants of sort `S`), the following alternatives are allowed:

(a) `X` = (empty) ; `Y` = ab
(b) `X` = a ; `Y` = b
(c) `X` = ab ; `Y` = (empty)

If we choose alternative (a), and a condition, associated with the equation, fails (possibly after many rewrite-steps), alternatives (b) and (c) still have to be considered. Clearly, a mechanism for backtracking is necessary here. For a detailed description of this mechanism, the reader is referred to [Wal91].

Next, the testing of conditions will be considered. In the case of conditions that do not contain 'new' variables (variables, not occurring in the left-hand-side of the equation), first both sides of the condition will be normalized. If the condition is an equality, it will succeed if and only if the resulting terms are equal. Otherwise, it succeeds if and only if the resulting normal forms are different.
Only one side of an equation is allowed to contain new variables in the current implementation of EQM. For such equations, the following strategy is adopted. First, the side, that does not contain new variables, is normalized. Then, the other side is matched against this normalized term, thereby defining bindings for the new variables. A requirement here, is that the side containing new variables is in normal form. Otherwise, the condition will always fail. For a complete description of the reduction strategy of EQM, the reader is referred to [Wal91].

Since the evaluation of conditions involves the normalization of its two sides, more than one (partially rewritten) term may exist during rewriting. The following situation is illustrated in Figure 15, below.

- A term is offered to EQM for rewriting. We call this first term the original term.
- When a subterm of the original term (indicated by shading) is considered, a match is found with conditional equation `EQ`.
- The left-hand-side of the condition of `EQ` is instantiated, resulting in term #2. Now, term #2 has to be normalized first, before work on the original term may continue.
- Currently, the subterm of term #2, that is indicated by shading, is considered for matching.

From this example we can see, that many, partially rewritten, terms may exist at each moment during rewriting.

The situation, that was sketched in the previous example, shows that every rewrite-step occurs at a definite level. Intuitively, this level denotes the current number of (nested) conditions. In the Equation Debugger, this notion of levels is used to skip the evaluation of conditions. This way, only rewrite-steps at the current level are regarded.

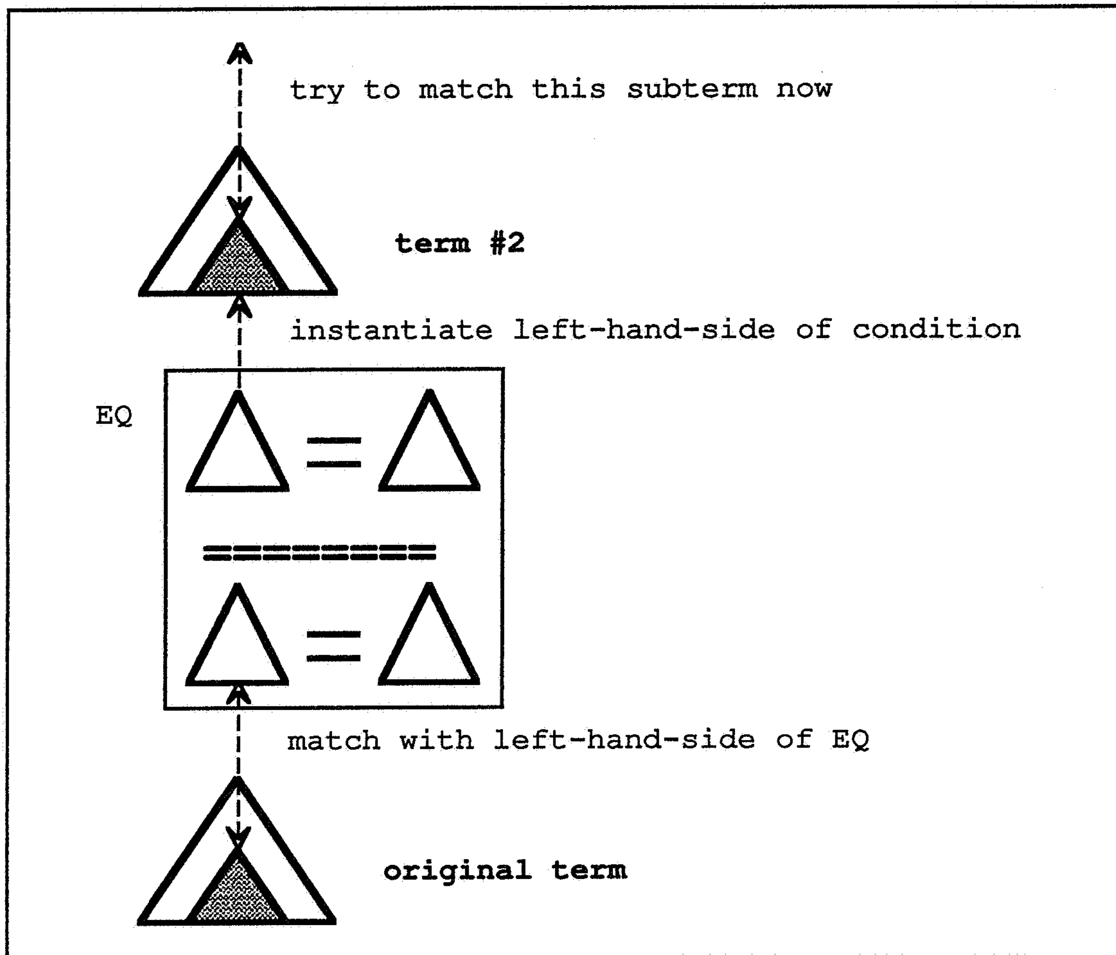


Figure 15. Multiple terms, a result of using conditional equations.

Next, the differences between innermost and outermost strategies for term-rewriting will be summarized. In the current implementation of EQM, a leftmost-innermost reduction-strategy is used. When an innermost strategy is adopted, first all subterms of a term are normalized, before the term itself is considered. In a leftmost-innermost strategy, the subterms of a given term are normalized in a strict order, i.e., from left to right. In the case of an outermost reduction strategy, first a match is searched for the complete term. If no match could be found there, the immediate subterms are considered, and so on.
The main advantage of an innermost reduction strategy occurs when a term is matched against the left-hand-side of an equation. In this case, all variables will be bound to terms, that are in normal form. This knowledge can be used, in order to prevent unnecessary matching. To see this, consider the case where a rewrite-step has just been performed. Now, the next subterm that is selected for matching will be (a subterm of) the result of the previous rewrite-step, due to the leftmost-innermost strategy. Because of the fact, that all variables are bound to terms in normal form, no (subterms of) bindings of the previous rewrite-step have to be considered for matching.
An important advantage of outermost strategies is, that no redundant work is done, in the case of non-strict functions, such as in the example below (see Figure 16). A non-strict function is a function, that does not always need all the values of its arguments. The `if-then-else` function below, is only strict in its first argument, i.e., the condition. Of course, this condition needs to be normalized first

in both reduction strategies, otherwise neither `If1` nor `If2` is applicable. If the leftmost-outermost reduction-strategy is used, `If1` or `If2` will be applied, before normalizing `Series1` or `Series2`. An innermost strategy would do redundant rewrite-steps, by first normalizing both `Series1` and `Series2`, before applying one of the equations below.

```
[If1] if true  then Series1 else Series2 = Series1
[If2] if false then Series1 else Series2 = Series2
```

Figure 16. Example of a non-strict function.

5. The functionality of the Equation Debugger

5.1 Introduction

The functionality of the Equation Debugger is described in this chapter. At the same time, the user-interface of EDB is presented. In Section 5.2, an overview is given of EDB's main objectives, and global design issues. Section 5.3 continues with the presentation of EDB's user-interface. In Section 5.4, the interface between the Equation Debugger, and the Equation Manager will be studied in detail. Finally, in Section 5.5, the evaluation of a term will be traced, using EDB.

5.2 Objectives and Design Issues

The main objectives of the Equation Debugger are twofold:

(1) The user should be provided with a powerful means for debugging the equations part of ASF + SDF specifications.
(2) Tracing, i.e., visualization of the rewrite-process should be supported. In the current situation, one only gets to see the original term, and its normal form. Often, more information is needed, in order to find bugs in a specification.

In order to meet the first objective, a variable execution step-size is needed. A second requirement is a flexible way of dealing with conditions. The user should be able to choose either to trace conditions step-by-step, or to skip conditions. In the latter case, all intermediate steps are skipped, and the final result of the condition, success or failure, is presented at once. Of course, information should be available about the conditions, that are currently being evaluated, during rewriting. Finally, some way of specifying breakpoints in the context of term-rewriting is necessary, to meet the first objective.
The second objective of EDB necessitates a way of displaying partially rewritten terms. Preferably these terms should be pretty-printed. Lastly, the current redex should be indicated in some way.

At this point, we will discuss the design issues of the Equation Debugger:

- Debugging is done interactively.

- The rewrite-process can only be observed, not influenced.

The first of these design issues says, that debugging is done during the rewriting of a term. At specific points, rewriting is suspended, and the user is given the opportunity of inspecting the term, setting breakpoints, and so on. We want to emphasize here the difference between interactive debuggers, as opposed to so-called post-mortem debuggers. As mentioned, interactive debugging takes place during execution (here: rewriting). Post-mortem debugging is a two-phase process. The first phase consists of gathering information during program execution. The second phase is the actual debugging, it consists of tracing the execution step-by-step, based on the information gathered in the first phase.

Next, we will discuss the second design-issue, i.e., the restriction of EDB to an observer of the rewriting process. The main reason for this restriction is, that a more powerful debugger is not feasible at present. It is not difficult to imagine a debugger, that allows the user to edit partially rewritten terms, or to change the reduction-strategy. However, a completely different interface with EQM would be necessary in that case. A simple master-slave interface, as we will describe in Section 5.4, where EQM passes information over to EDB, would no longer be sufficient. Instead, a much more complicated interface would be required, with information flowing in two directions between EQM and EDB. Unfortunately, such a bi-directional interface necessitates a new implementation of EQM.

5.3 The user-interface of EDB

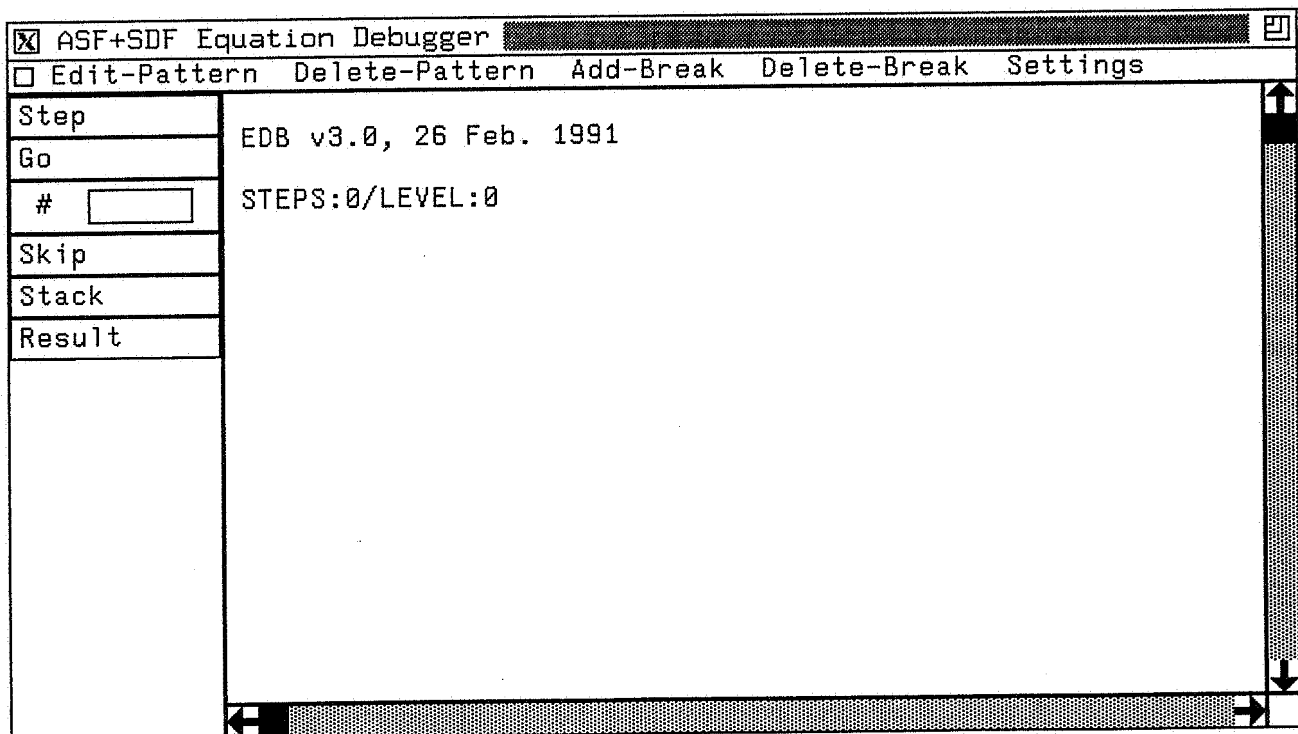


Figure 17. The main window of EDB.

In this section, the user-interface of the Equation Debugger is presented. In Figure 17, above, the main window of EDB is shown, which consists of the following three parts:

- A scrollable output-window, used for displaying information about the rewriting process.
- A column of buttons for the most often used commands.
- A row of menus, providing items for manipulating breakpoints, changing the current settings, and terminating the debug-session.

In Figure 17, we see that after start-up of EDB, the message `STEPS:0/LEVEL:0` is displayed. This message indicates, that no rewrite-steps have been done yet, and that the current condition-level is 0. Generally, the message `STEP:`*n*`/LEVEL:`*l* is printed every time, that EDB hands over control to the user. Here, *n* is the number of rewrite-steps done so far, and *l* denotes the current level.
The amount of other information, that is displayed in EDB's main window depends on the current settings. For example, the user may choose whether or not to show the equations, that are applied. The settings of EDB will be commented upon shortly.
The partially rewritten term is not displayed in EDB's main window, because printing all these terms would result in too much text presented to the user. Instead, an instance of GSE is used for pretty-printing partially rewritten terms. In this GSE-instance, the focus is positioned on the current redex. In Figure 18, below, EDB's window for pretty-printing is presented.

```
EDB pretty
tree text expand help
ev
[
  let
    a = succ succ 0,
    c = 0,
    b = succ pred pred 0,
    cc = 0
  in
    ( a + b * let
                a = succ succ succ succ 0
              in
                a + const
                    (
                      succ succ 0
                    ) )
]
```

Figure 18. EDB's window for pretty-printing.

The communication between EDB and the user is roughly as follows:

(a) EDB gets a command from the user.
(b) The command is executed.

(c) If (b) did not involve any rewriting, we return to (a). Otherwise, information about the rewriting process is displayed in both windows described above. As mentioned before, the amount of information displayed depends on the current settings.

(d) If normal form is reached, the user is informed about this fact, and the total number of rewrite-steps is displayed. Otherwise, the `STEP:`*n*`/LEVEL:`*l* message is displayed, and we return to (a).

At this point, we will sketch the functionality provided by the buttons of the main window of EDB.

- `step`
  When this button is selected, the rewriting process will continue until either a rewrite-step is performed, *or* a change in the current level occurs. This mode of operation of EDB is called *step-mode*. Steps done in step-mode are the smallest possible steps of EDB.

- `go`
  Associated with the `go`-button is the editable field labeled '#' below it. When the `go`-button is pressed, and the #-field is empty, the rewriting process continues until either normal form is reached, or a breakpoint is encountered. This mode of operation of EDB is called *go-mode*.
  In the case that a positive integer n is entered in the #-field, the *extra* stop-condition applies, that no more than n rewrite-steps should be performed. In other words, control is returned to the user after at most n rewrite-steps.
  If the #-field contains a string that is not a positive integer, pressing the `go`-button results in the message '`go: number of steps not correct.`' being displayed. In that case, control is returned to the user immediately.

- `skip`
  If conditions are currently being evaluated, pressing the `skip`-button will continue the rewriting process, until the condition-level is one less than the current level. If meanwhile a breakpoint is encountered, control is returned to the user. However, in that case the `skip`-action can be resumed, by pressing the `go`-button.
  If the `skip`-button is selected when no condition is under evaluation, the message '`skip: current level is 0.`' is displayed, and control is returned to the user immediately.

- `stack`
  If conditions are under evaluation, EDB's current stack of conditions is displayed in the main window upon selection of the `stack`-button. As described in Chapter 4, the use of conditional equations causes the co-existence of multiple terms. For each of these terms, the tag of the matching equation is displayed, followed by the current evaluation-status of the conditions of the matching equation.
  In the case that the `stack`-button is pressed, and no conditions are being evaluated, the message '`stack: condition stack is empty.`' is displayed.

- `result`
  Due to an optimization in the Equation Manager (which will be discussed in Section 5.4), the result of a rewrite-step generally is not available immediately after applying the equation. Instead, other rewrite-steps may take place in the meantime.
  By pressing the `result`-button these 'intermediate' rewrite-steps are skipped, i.e., rewriting continues until the result of the last rewrite-step becomes available. If a breakpoint is encountered meanwhile, control is returned to the user. As in the case of the `skip`-button, the `result`-action can be resumed after the breakpoint, by pressing the `go`-button.
  Selection of the `result`-button at the start of the rewriting, when no rewrite-steps have been done yet, results in the message '`result: no steps done yet.`'. After this, control is returned to the user.

Before we will describe the functionality of the menus of the main window of EDB, we will examine the use of breakpoints. EDB provides three kinds of breakpoints:

(1) *Break-patterns* are terms, which are matched with the current redex, whenever rewriting is done in go-mode. When such a match succeeds, rewriting is interrupted, and the message '`*`n`* break at` $p = t$' is displayed. Here, n is the number of rewrite-steps done so far, p is the name of the pattern that matched with the redex, and t is the (first line of the) actual text of the pattern. After presenting this message, information about the current rewrite-step is displayed, and control is given to the user.

(2) *Display-patterns* are actually a variant of break-patterns. If a match with a display-pattern succeeds, while rewriting in go-mode, information about the current rewrite-step is displayed, but rewriting continues. In the case of display-patterns, the slightly different message '`*`n`* display point at` $p = t$' is presented to the user (n,p, and t as above). The main purpose of display-points is to inform the user about the current situation of the rewriting.

(3) Finally, the *tag* of an equation can be specified as a breakpoint. When rewriting is performed in go-mode, and an equation is being applied, with a tag that was designated as a breakpoint, rewriting is suspended. In this case, the message '`*`n`* break at` x' is displayed, where n is the number of rewrite-steps done so far, and x is the tag on which the breakpoint was set. After this message, information about the rewriting is printed, and control is given to the user.

The menubar of the EDB-window contains the button □, and the menus `Edit-Pattern`, `Delete-Pattern`, `Add-Break`, `Delete-Break`, and `Settings`. Now, we will describe the functionality provided by these options.

- When the □ button is pressed, the Equation Debugger is left, and the rewriting process is terminated.

- The `Edit-Pattern` menu contains items for the creation and modification of break- and display-patterns.

When the `New` item is selected, a dialog pops up, asking the user to enter the filename of the pattern (see Figure 19). If the `Cancel` button of this dialog is pressed, the command is terminated. After selection of the `OK` button, a file is searched with the filename entered by the user.

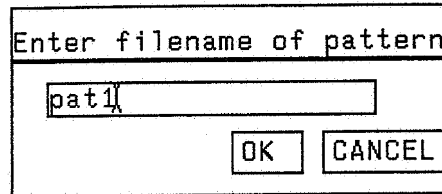


Figure 19. Dialog asking for filename of pattern.

If a file with the appropriate filename can be found, its contents are pasted in a pattern-editor. We will discuss pattern-editors shortly. Otherwise, the user is confronted with a new dialog, asking whether a new pattern should be created (see Figure 20). Again, selection of the `Cancel` button terminates the command. Pressing the `OK` button of the dialog below results in starting an empty pattern-editor.

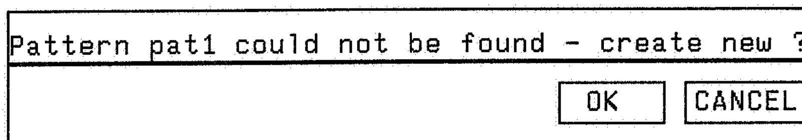


Figure 20. Dialog asking if a new pattern should be created.

Figure 21 below shows a pattern-editor. *Pattern-editors* are GSE-instances, that are used for editing break- and display-points. These GSE-instances are parameterized by the language of the term-editor that supplied the original term, extended with all hidden syntax.

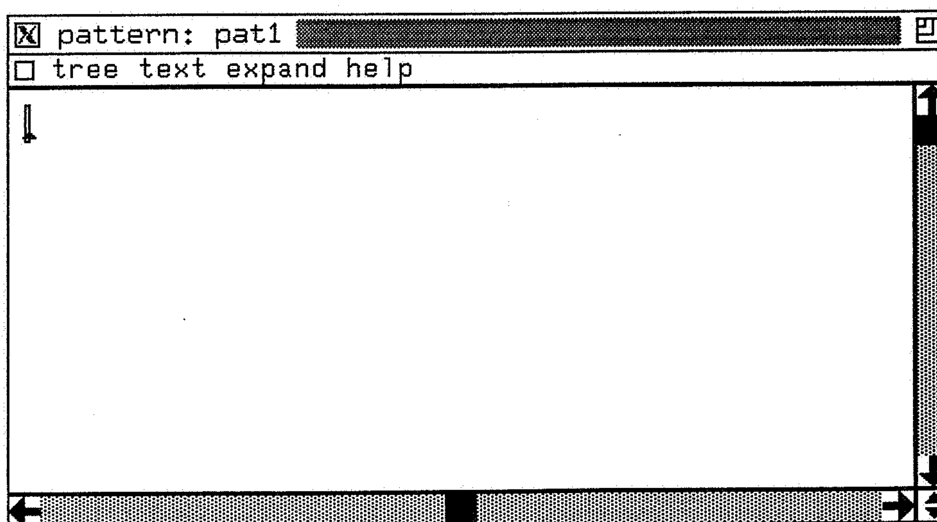


Figure 21. A pattern-editor.

When the editing of the pattern is finished, the □ button of the pattern-editor should be clicked. If the pattern that was entered, is syntactically correct, this will result in presenting the dialog of Figure 22 to the user, with the following alternatives:

(1)	`Set Break`	:	The pattern in the editor is designated as a break-pattern, and the editor is left.
(2)	`Set Display`	:	The pattern in the editor is designated as a display-pattern, and the editor is left.
(3)	`Trash`	:	All editing done by the user is discarded, and the editor is left.
(4)	`Cancel`	:	The user may continue editing (the dialog disappears).

If the text in the pattern-editor is not a syntactically correct term, only the options `Trash` and `Cancel` will be present in the dialog.

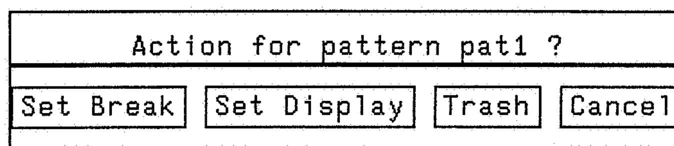
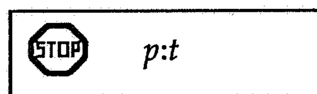
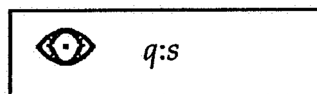


Figure 22. Dialog for entering patterns.

The other items of the Edit-Pattern menu correspond to the patterns, that were entered previously. For every pattern that was entered, an item will be included, containing the name, and the first line of text of that pattern. The type of the pattern (break or display) is expressed by an icon in the item:
For a previously entered break-pattern with name p and (first line of) text t, the following item is included:



For a display-point-pattern with name q and (first line of) text `s`, the following item is included:



After selection of one of these items, a pattern-editor will be started with the text of the corresponding pattern.

- The `Delete-Break` menu provides items for the deletion of breakpoints on tags of equations. After selecting the first item in this menu, `All`, a dialog asks the user to confirm the deletion of all breakpoints on tags. Here, a similar dialog as in Figure 23 is used.
The other items of the `Delete-Break` menu are labeled with the tags, that were previously entered. Here, the same kind of item is used for previously

entered patterns, as in the `Edit-Pattern` menu. Selecting one of these items removes the corresponding breakpoint.

- Finally, the `Settings` menu allows the user to control the amount of information, that is shown to the user. Associated with each item in this menu is a 'check-box'. This check-box indicates whether or not the corresponding setting is selected, by way of the icons ☐ and ☑. Below, the items of the `Settings` menu are listed.

  - `Print Bindings`
    Display the bindings, that are used when an equation is applied.
  - `Print Equations`
    Display the applied equations.
  - `Print Results`
    Display the results of rewrite-steps, i.e., the instantiated right-hand-sides of the equations.
  - `Instantiate Conditions`
    Determines the format of printing conditions. When selected, conditions are displayed after instantiation. Otherwise, conditions are shown uninstantiated, together with the current bindings.
  - `Show Tags`
    Determines whether or not tags of applied equations should be printed, when rewriting is done in go-mode.
  - `Pretty-print lv.0 only`
    Determines when the current term is pretty-printed in EDB's window for pretty-printing. When selected, the window is only updated when the current level is 0. Otherwise, terms at all levels are pretty-printed.
  - `Stop at breaks`
    Determines whether or not break- and display-patterns, and breakpoints at tags are effective.

5.4 The interface between EDB and EQM

In this section, the communication between the Equation Debugger, and the Equation Manager is defined. Through the interface, that will be described shortly, EQM informs EDB of the current status of the rewrite-process. This interface consists of 10 interface-functions in EDB, called by EQM at significant points during the term-rewriting process. (Such as, the start of the rewriting of a term.) A short overview is given, of the actions taken by EDB, when one of these interface-functions is called by EQM.

Whenever a term is going to be rewritten, and debugging is required, a new *instance* of EDB is initiated. Each instance of EDB has its own window, and maintains its own set of values. The most important of these values are:

- The stack of terms, containing a copy of the (partially rewritten) terms, that are currently being evaluated.
- The stack of conditions. This stack contains information about conditions of the equations, that matched with the terms on the stack of terms.
- The current level.

- The number of rewrite-steps done, so far.
- Various flags for the current settings.

We will now list the 10 interface-functions EQM uses, to pass status-information to EDB. For each of these functions, a description is given of:
- when the function is called.
- the arguments of the function.
- actions taken by EDB as a result.

`reduce-start (EQMsel, term)`
- This function is called once, when a term is offered to EQM to be rewritten.
- `reduce-start` has two arguments. `EQMsel` indicates the current selection of equations. The selection-mechanism, that is used in the ASF + SDF system, was sketched briefly in Section 3.6. The other argument, `term`, is the original term, that is going to be rewritten.
- As a result of this function call, a new instance of EDB is created, and a window is created for that new instance. Furthermore, `term` is pretty-printed in a separate window, and EDB is initialized. The user can change the default settings, or commence the rewrite process.

`reduce-end (EQMsel, term, result)`
- `reduce-end` is called once, when rewriting is finished, and normal form is reached.
- The first two arguments of `reduce-end` are the same as for `reduce-start`. The third argument, `result`, is the final result of the rewriting, i.e., the normal form.
- In response to a call to `reduce-end`, EDB pretty-prints `result`, and displays the total number of rewrite-steps executed.

`match (EQMsel, term, eq, bindings)`
- The function `match` is called, whenever a match is found, given a term and an equation.
- The first argument, `EQMsel`, is the current selection of EQM. The arguments `term` and `eq` are the subterm and equation involved in the match, respectively. Finally, `bindings` is a list of (variable,value)-pairs, that contains the bindings found during matching.
- The only action taken by EDB, is that a marker is pushed on the condition-stack, indicating that a new equation is considered.

`eval-cond-start (EQMsel, cond, eq, bindings)`
- `eval-cond-start` is called by EQM, before evaluating a condition.
- Again, `EQMsel` is the current EQM-selection. `eq` is the equation, to which condition `cond` belongs. The last argument, `bindings`, consists of the bindings found so far, during matching and in previous conditions.
- EDB pushes the condition on the condition-stack, and the current level is incremented by one. Depending on the settings, the condition is displayed in EDB's window. If the current settings indicate that conditions should be traced step-by-step, the user may alter the settings, enter breakpoints, or give other commands. Otherwise, rewriting continues without intervention by the user.

eval-cond-end (EQMsel, cond, eq, bindings)

- This function is called after the evaluation of a condition.
- The arguments EQMsel, cond, and eq are the same, as for eval-cond-start. If the condition succeeded, bindings contains the possibly extended list of bindings (new variables may have been introduced). In case of failure of the condition, bindings gets the special value **nil**.
- The condition-stack is updated, and the current level is decremented by one. Depending on the current settings, the condition and bindings are displayed. Also, the user may alter the settings, enter breakpoints, or give other commands. If the current debug-mode was go-mode, and this condition was skipped, EDB returns to step-mode.

cond-side-start (side, bindings)

- This function is called, whenever a side of a condition is going to be normalized.
- The first argument, side, is a term representing the current condition-side. The other argument, bindings, is the list of bindings, established so far.
- A copy of side is instantiated, and pushed on EDB's stack of terms.

cond-side-end (side)

- cond-side-end is called, when a condition-side has been normalized.
- The only argument, side, is the current condition-side.
- EDB's term-stack is popped.

apply-eq-start (EQMsel, term, eq, bindings)

- The function apply-eq-start is called, after a match was found, and all conditions succeeded.
- EQMsel is the current selection of EQM. The arguments term and eq are the term and equation involved in the match. The last argument is a list containing all bindings to be used for instantiating the right-hand-side of the equation.
- The number of rewrite-steps performed is increased by one. All information regarding conditions of eq, is popped from the condition-stack. The subtree, corresponding to term on EDB's stack of terms is replaced by the instantiated right-hand-side of eq. If the current debug-mode is step-mode, the user is allowed to alter settings, set breakpoints, and give other commands. If EDB operates in go-mode, a test is performed whether a breakpoint is encountered. If so, the user is given control.

apply-eq-end (EQMsel, term, eq, result)

- This function is called after the application of an equation.
- The first three arguments are the same as those of apply-eq-start. The last argument, result, is the result of the rewrite-step.
- Depending on the current settings, the result of the rewrite-step is displayed.

noapply-eq (EQMsel, term, eq)

- If, after a match, some condition failed, noapply-eq is called.
- The three arguments of noapply-eq denote the current selection of EQM, and the term and equation under consideration, respectively.

- All information regarding conditions of `eq`, is popped from the condition-stack.

In some cases, the Equation Debugger requests services from the Equation Manager:

- A function `instantiate (term, bindings)` is called, when EDB needs to instantiate the right-hand-side of an equation, given a list of bindings. The reason for this, is that EDB constructs its own private copy of the (partially rewritten) term(s) of EQM.
- A function `match (EQMsel, term, pattern)` is called, when EDB needs to test, if a given subterm `term` matches with breakpoint `pattern`.
- The function `pretty (term)` is called for pretty-printing terms.
- By raising the exception `abort`, EDB can give a signal to EQM, that the rewriting process should be terminated.

In the current implementation of EQM, an important optimization is included. After each rewrite-step, a recursive call to EQM's `rewrite` function puts the subterm, that forms the result of that rewrite-step, in its normal form. Only after normalization of this result, the rewriting of the complete term continues. As a result of this optimization, calls to `apply-eq-start`, and the corresponding `apply-eq-end`, do not have to occur consecutively, since other rewrite-steps may have taken place in the time between these two function calls.
Unfortunately, this seriously affects the implementation of EDB. The problem is that the complete term usually is not available, and at some point a subterm may not have a parent, although formally it should have one.
The solution for this problem, used in the present implementation, is that EDB constructs its own copy of all terms that are to be rewritten. Hence, EQM is used merely for indicating which redexes are to be replaced by which results. We are aware that this solution is inefficient, and thus not very satisfactory. However, no alternative could be devised so far. In fact, we feel that providing the complete status of the rewrite-process, including all partially rewritten terms, and conditions under evaluation, should be part of the functionality of a future implementation of EQM.

5.5 An example: tracing the evaluation of a term

In this section, a small example will be presented, in which the Equation Debugger is used to trace the rewriting of a term. In our example here, the term presented earlier in Figure 2, will be evaluated using the `ev` function of module `Exp-ev` (see Figure 9). Figure 23 below shows this term in EDB's window for pretty-printing, when EDB is started.

Next, we will describe the actions of the user, and the corresponding output in the main window of EDB, which is presented below, in Figure 29 and Figure 30.

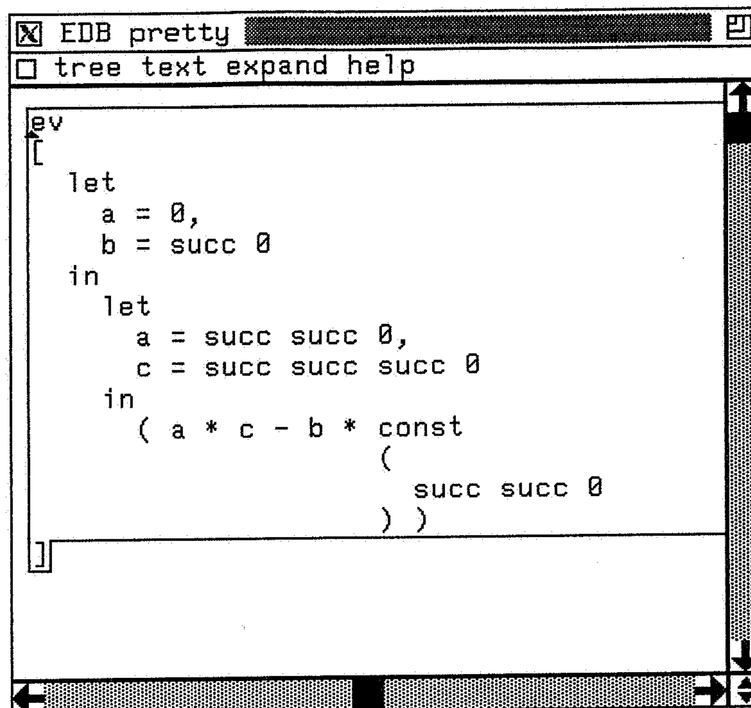


Figure 23. The original term in EDB's window for pretty-printing.

- After the start-up message `STEPS:0/LEVEL:0`, the user issued a `step` command. As a result, rewriting started, and control was returned to the user, because a change in condition-level occurred.
  Observe, that this change in condition-level was caused by starting the evaluation of the condition of equation `Ev1` of module `Exp-ev` (Figure 9):

```
[Ev1]      tc[ E ] = true
           ======================
           ev[ E ] = [ E ] in ()
```

  Obviously, the current settings determine that conditions were displayed in instantiated form, since the variable `E` in the equation is replaced by the complete expression in the output.
  After displaying the message `STEPS:0/LEVEL:1`, control was returned to the user.
- At this moment, the user was not interested in the individual rewrite-steps of the condition. Therefore, the `skip` command was executed. Rewriting continued until the condition was fully evaluated. We see in the output, that the condition mentioned above, succeeded. After presenting the message `STEPS:34/LEVEL:0`, control was returned to the user. This means, that the evaluation of the condition required 34 rewrite-steps.
- Next, two breakpoints were entered. A break-pattern with name `pat1`, and text `sub <INT> <INT>` was defined first, followed by a break on equation `Ev3`. After this, the user indicated that tags of applied equations should displayed during rewriting in go-mode, and then the `go` command was issued.
- From the tags printed in the output we see, that many equations were applied until a breakpoint was encountered at the 66th rewrite-step. After

signalling this break, the current equation and bindings were displayed, followed by the message `STEPS:66/LEVEL:0`.
Figure 24 shows the term, that was displayed in EDB's window for pretty-printing at that moment.

```
EDB pretty
tree text expand help

sub succ succ succ succ succ succ 0 mul succ 0 [
                                                 const
                                                 (
                                                   succ succ 0
                                                 )
                                               ]
                                               in
                                                 (
                                                   c : succ succ succ 0
                                                   a : succ succ 0
                                                   b : succ 0
                                                   a : 0
                                                 )
```

Figure 24. Partially rewritten term at rewrite-step #66.

- Another `go` command was executed hereafter. Observe, that more equations were applied, until a breakpoint was encountered at the 74th rewrite-step. Here, a match with pattern `pat1` (`sub <INT> <INT>`) was detected, when equation `Int7` was applied:

  `[Int7] sub Int0 succ Int1 = sub pred Int0 Int1`

  Again, the current equation and bindings were printed, this time followed by `STEPS:74/LEVEL:0`. Figure 25 shows the contents of EDB's pretty-print window at that time, with the focus positioned on the redex.

```
EDB pretty
tree text expand help

sub succ succ succ succ succ succ 0 succ succ 0
```

Figure 25. Partially rewritten term at rewrite-step #74.

- We see, that the same break-pattern, `pat1`, matched twice again with a redex. The first time at the 76th rewrite-step, when equation `Int7` was applied again (Figure 26 shows the term at that time).

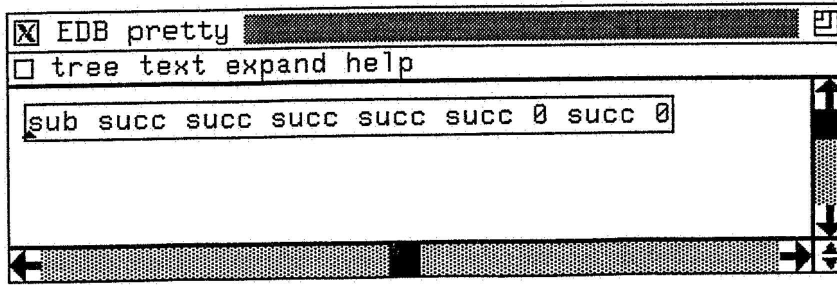


Figure 26. Partially rewritten term at rewrite-step #76.

The second time, `pat1` matched with the left-hand-side of equation `Int6`, at the 78th rewrite-step.

```
[Int6] sub Int 0 = Int
```

Figure 27 below shows the term at that time.

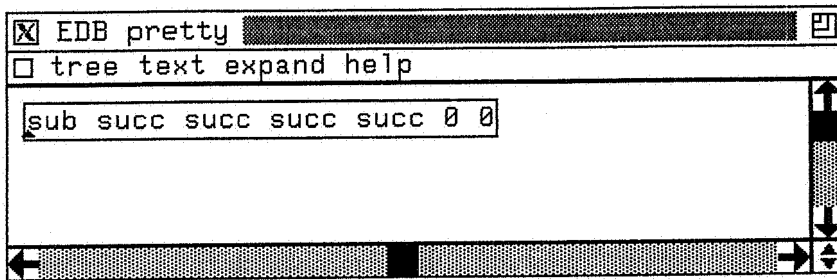


Figure 27. Partially rewritten term at rewrite-step #78.

- After 78 rewrite-steps, the term was normalized. The normal form is displayed in EDB's pretty-print window in Figure 28.

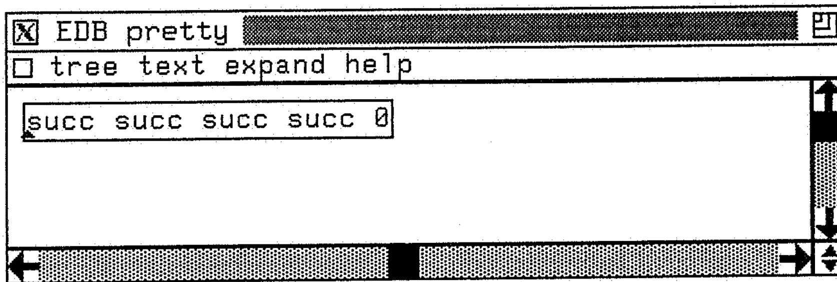


Figure 28. Normal form of the original term.

```
[X] ASF+SDF Equation Debugger
[ ] Edit-Pattern  Delete-Pattern  Add-Break  Delete-Break  Settings

Step
Go
#
Skip
Stack
Result

 EDB v3.0, 26 Feb. 1991

 STEPS:0/LEVEL:0
   ``tc [ let a = 0, b = succ 0 in let a = succ succ 0, c =
succ succ succ 0 in a * c - b * const ( succ succ 0 ) ] = tr
ue'' for equation [Ev1] ???
 STEPS:0/LEVEL:1
   ``tc [ let a = 0, b = succ 0 in let a = succ succ 0, c =
succ succ succ 0 in a * c - b * const ( succ succ 0 ) ] = tr
ue'' for equation [Ev1] SUCCEEDED.
 STEPS:34/LEVEL:0
   Break at pat1 = sub <INT> <INT> added.
   Break at Ev3 added.
   Ev1
   Ev2
   Ev10
   Ev10
   Ev9
   Ev2
   Ev10
   Ev10
   Ev9
   Ev7
   Ev8
   Ev5
   Ev4
   Ev4
   Int10
   Int10
   Int10
   Int9
   Int4
   Int4
   Int3
   Int4
   Int4
   Int3
   Int4
   Int4
   Int3
   Ev8
   Ev5
   Ev5
   Ev4
   *66* break at Ev3
   EQUATION:
     [Ev3]
     [ const ( Int ) ] in Venv =
           Int
```

Figure 29. First part of the output in EDB's main window.

```
ASF+SDF Equation Debugger
Edit-Pattern  Delete-Pattern  Add-Break  Delete-Break  Settings

Step
Go
#
Skip
Stack
Result

  BINDINGS:
    Venv =
      ( c : succ succ succ 0 a : succ succ 0 b : succ 0 a :
0 )
    Int =
      succ succ 0
STEPS:66/LEVEL:0
  Int10
  Int10
  Int9
  Int4
  Int3
  Int4
  Int3
  *74* break at pat1 = sub <INT> <INT>
  EQUATION:
    [Int7]
    sub Int0 succ Int1 =
              sub pred Int0 Int1
  BINDINGS:
    Int1 =
      succ 0
    Int0 =
      succ succ succ succ succ succ 0
STEPS:74/LEVEL:0
  Int2
  *76* break at pat1 = sub <INT> <INT>
  EQUATION:
    [Int7]
    sub Int0 succ Int1 =
              sub pred Int0 Int1
  BINDINGS:
    Int1 =
      0
    Int0 =
      succ succ succ succ succ 0
STEPS:76/LEVEL:0
  Int2
  *78* break at pat1 = sub <INT> <INT>
  EQUATION:
    [Int6]
    sub Int 0 =
              Int
  BINDINGS:
    Int =
      succ succ succ succ 0
STEPS:78/LEVEL:0

  End of reduction, total number of rewrite-steps : 78.
```

Figure 30. Second part of the output in EDB's main window.

6. Debugging term rewriting systems compared with debugging conventional programs

A comparison between debugging techniques for term rewriting systems, and for conventional programs will be given in this chapter. Below, we will present this comparison, using a list of desired features of debuggers for high-level programming languages as a guide-line. Such a list of desired features can for example be found in [Fer83]. For each of the features discussed below, the following three cases are considered:

(1) A debugger for an imperative language as, for example, Pascal.
(2) A debugger for Prolog (we will use C-Prolog's debugger for the example).
(3) A debugger for a term rewriting system.

The choice of the smallest execution step-size

In debuggers for imperative languages, the smallest step-size may vary from expressions to statements, together with the entry and exit of subprograms (i.e., procedures and functions).

In C-Prolog's debugger, the so-called Procedure Box model is used to express the control flow. In this model, a procedure consists of all clauses of a predicate. A procedure box has four ports. The `Call` port represents the initial invocation of the procedure. The second port, `Exit`, represents a successful return from the procedure. The `Fail` port expresses failure of the initial goal. Lastly, the `BackTo` port is entered, when a subsequent goal has failed, and backtracking results in trying another clause of the procedure. A full description of this model can be found in [Prolog86]. The smallest step-size consists of a transition between ports.

In a term rewriting system, a single rewrite-step could be selected as the smallest step. However, since rewrite-rules may have conditions, it becomes necessary to define the begin and end of the evaluation of a condition to be a step of the debugger as well. Otherwise, the user would become confused, not knowing which term is currently being rewritten.

As a matter of fact, an even smaller step would be acquired, by defining every match of a given term and equation to be a step, and/or every begin and end of the evaluation of a condition-side. However, some experiments during the implementation of EDB revealed, that defining these two events to be a step would not add much relevant information. Instead, many unwanted steps would be added. The reason for not defining every match to be a step of the debugger, is that on entering and leaving conditions, the equation involved in the match is already mentioned. The reason for not defining every begin and end of the evaluation of a condition-side to be a step, is that nearly always one side of a condition is in normal form.

A level notion in debuggers

Debuggers for imperative languages associate a concept of level with the nesting of subprogram-calls. Consequently, levels can be traced step-by-step, or skipped completely.

In the C-Prolog debugger, a level notion is connected to the entering and leaving of procedures. Backtracking complicates this notion, since procedure

boxes can not only be entered and left, but also re-entered. Again, the user can choose either to step through procedures, or to skip procedures entirely.
The level concept in term rewriting systems was described earlier. It is based on the nesting of the evaluation of conditions.

Breakpoints

Debuggers for imperative languages allow the user to set breakpoints on a statement, or on a line of source code. When program execution reaches the compiled code corresponding to that statement, or line of source-code, the user is given control.
The C-Prolog debugger allows the user to set breakpoints (called spy points) on predicates.
In debuggers for term rewriting systems, breakpoints can be set on both patterns occurring in the term, or on rewrite-rules. Presently, EDB supports patterns, that are matched against the current redex. However, one could also envisage patterns to be matched with results of rewrite-steps, or with the (complete) terms itself.
Because of the fact that patterns can be as general as a single meta-variable, or as specific as any syntactically correct term, the use of patterns as breakpoints seems much more powerful than the breakpoints used in programming languages.

Inspection of variables and source-level expressions

One of the most often used features of a debugger for an imperative language, is the inspection of the current value of a variable. Generally, the value of more complex source-level expressions can often be inspected too.
In the case of C-Prolog's debugger, Prolog's database can be inspected during debugging. More complex expressions can be inspected during debugging by trying the appropriate goals.
When in debuggers for term rewriting systems a program is evaluated, the term being rewritten contains both the partially rewritten program text, and the variables together with their current values as subterms.
This means that the inspection of variables is just a special case of the inspection of subterms of the complete term. For displaying more complex source-level expressions, no such analogy exists.

Modification of the values of variables, and modification of the program

In the case of debuggers for imperative languages, these are two different issues. Modifying the value of a variable corresponds to putting a new value in the memory-location of that variable. Modifying the program itself during execution is something quite different. A new piece of source-code will have to be compiled, and the resulting code should be substituted for the corresponding old piece of code.
If we consider the C-Prolog debugger, the prolog-database may be changed during debugging. This can of course lead to unexpected behaviour, when backtracking re-enters a procedure-box, to which predicates have been added, or from which predicates have been deleted.
In debuggers for term rewriting systems, modifying variables and modifying the program both correspond to changing the current term. For reasons

mentioned earlier, modification of the current term is currently not implemented in EDB. By changing the language-definition that is used during rewriting, execution behaviour could also be influenced.

7. Concluding remarks

A specification written in the formalism ASF + SDF simultaneously defines the syntax and the semantics of a language. The semantics of a language are expressed in a set of conditional equations. In the ASF + SDF system, the set of equations in a specification is regarded as a term rewriting system.
In this report, we have presented the Equation Debugger, a highly interactive debugger for the ASF + SDF system. The most significant feature of EDB is a flexible mechanism for specifying breakpoints, based on patterns to be matched against the current redex. These patterns may vary from being very general to being very specific. In this way, the user may trace the rewriting process with an execution step of variable size.
At any time that the user has control, the amount of information about the rewriting process that is displayed, may be adjusted. Furthermore, the tracing of the evaluation of conditions can either be done step-by-step, or it can be skipped. By using a separate window for displaying pretty-printed terms, the user is provided with the current status of the rewriting process.

The Equation Debugger has been implemented as part of the ASF + SDF system. Given the status of EDB as described in this report, one can already see a number of potentially interesting future extensions:

- Currently, EDB is restricted to being an observer of the rewriting process. In fact, adding functionality for changing terms during rewriting would require little work. However, a more complicated interface with the Equation Manager, and therefore a new implementation of EQM would be required. In this new situation, EDB would have to direct the rewriting that is performed by EQM.
- Since uniqueness of tags is not demanded in the ASF + SDF formalism, breakpoints should be set by interactively pointing at equations, rather than by entering tags.
- EDB is an interactive tool for debugging sets of equations. In the context of the generation of programming environments for specific languages, it would be interesting to provide generic mechanisms for creating language-specific debuggers. Further research could investigate if steps of such language-specific debuggers can be expressed in terms of steps of EDB.

Acknowledgements

I would like to thank Pum Walters and Paul Klint for many helpful comments on previous versions of this report, Wilco Koorn for explaining the use of the graphical toolkit and GSE, and Casper Dik for providing the interface with EQM.

References

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

[Centaur] The CENTAUR Documentation, The GIPE Consortium, Sophia-Antipolis.

[Hen91] P.R.H. Hendriks, *Implementation of Modular Algebraic Specifications*, dissertation, Programming Research Group, University of Amsterdam (1991), to appear.

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

[Fer83] J.Ferrante, "High level language debugging with a compiler", Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, pp.123-129 (1983).

[Kli90] P.Klint, "A meta-environment for generating programming environments", Report CS-9064, Centre for Mathematics and Computer Science, Amsterdam, 1990.

[Koo91] J.W.C. Koorn, "GSE: A generic text and structure editor", Programming Research Group, University of Amsterdam (1991), to appear.

[LeLisp87] LeLisp, Version 15.21, le manuel de référence, INRIA, Rocquencourt, 1987.

[Prolog86] C-Prolog User's Manual Version 1.4d.edai, Edited by F.Pereira, SRI International, Menlo Park, California, 1986.

[Wal91] H.R. Walters, *On Equal terms*, dissertation, Chapter 1: "An Equation Manager", Programming Research Group, University of Amsterdam (1991), to appear.