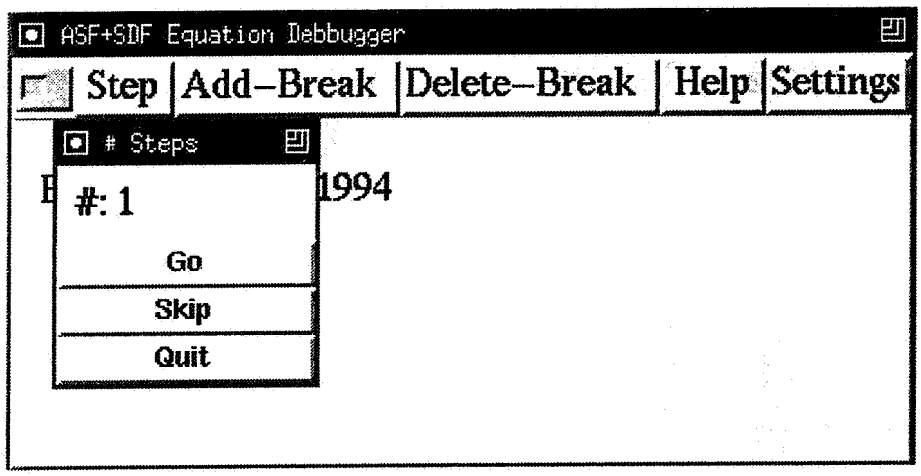
Equational Debugger
An Experimental Debugger for the ASF+SDF Term Rewriting System



J. N. Sappore-Siaw
Email: sappore@fwi.uva.nl

University of Amsterdam
Faculty of Mathematics and Computer Science
Master's Thesis

August 1995

Abstract

This thesis deals particularly with an experimental postmortem debugger, EDB, developed for the ASF+SDF term rewriting system. It has been found desirable to have a debugger that does not run concurrently with the ASF+SDF system. This implies that for a specification to be debugged, the rewrite process must have run through to the end and collected enough information which will be used by the debugger. Only after this information is collected can the debugger function commence.

Acknowledgements

I would like to thank Prof. Dr. Paul. Klint, for his support and guidance in bringing this thesis to pass. I will also very much like to express my special thanks to Dr. Frank Tip, my mentor. Dr. Wilco Koorn was especially helpful at the initial stages when everything was unclear - thank goodness Wilco's office was just next-door.

I would like to thank Frans van der Vliet for introducing me to computing. I am especially thankful to Suneel Shivdasani and Laura de Groot for their financial support, and otherwise.

My mates have to endure a lot of hardship in encouraging and challenging me to go on. I would also like to thank them all here by naming them: Fred Bonsu, Seyyed Kamal Koochak, Drona Kandhai and Mike Tai.

Contents

1 The ASF+SDF system

Before describing the debugger itself, the rest of this section will briefly introduce the ASF+SDF system - the environment in which the debugger is expected to run. Being postmortem, however, the debugger does not actually run in the ASF+SDF environment. But the interaction is such that one should not notice it is a separate program.

Except when explicitly stated, the debugger referred to in this section is the current version and is not the new version that this thesis is about.

1.1 A Bird-Eye View Of ASF+SDF

ASF+SDF is an integrated software system for the automatic generation of programming environments for formal language specifications. ASF+SDF is a highly interactive software system. It allows the editing of specifications for which it generates a programming environment. The environment consists of syntax-directed editor, type checkers, program evaluators and many other tools. A number of components can be distinguished in the ASF+SDF system; these include the :

- Syntax Manager
  which is responsible for the syntax.
- Equation Manager
  handles the rewriting process.
- Module Manager
  handles the modular structure defined in the specification.
- Generic Syntax-directed Editor
  rather instances of it, enable the edition of terms and specifications so that textual editions are reflected in the structure of the term or specification.
- Supervisor
  drives all other components of the system and interprets user-commands.
- Equation Debugger
  enables interactive debugging of a specification and provides a trace to the rewrite process.

A specification, for this purpose, has two parts; a syntax definition and a semantic definition. These are respectively the SDF and the ASF of the specification.

1.2 SDF - The Syntax Definition Formalism

The SDF part of a specification defines both the abstract and concrete syntax of the formal language using the formalism described by [HHKR89]. Both abstract and concrete syntax are expressed by regular and context-free grammar rules. A fixed mapping is defined from a syntax combination in the concrete syntax to another syntax in the abstract syntax. This mapping makes the definition of function overloading, functions, list, etc, a matter of course.

Terms that correspond to more than one abstract syntax tree are said to be ambiguous; ambiguity can be resolved by expressing priority and associativity on such a grammar.

Associativity is expressed as left-, right- or non-associativity. There is also the bracket directive that transforms the concrete syntax of a bracketed term into its abstract syntax representation.

Language definitions are modular and a definition generally consist of a set of modules. This modularity, in part, allows a degree of object hiding by using the hidden directive. A specification can also import the definitions of another specification; but the hidden part of the syntax can not be used by an importing specification.

Variables to be used in the semantic part of the specification must be declared in the SDF part, the type (sort, in ASF+SDF terminology) of which has been previously declared in the current or an imported module.

1.3 ASF - The Algebraic Specification Formalism

The ASF part of a specification defines the semantic definition of the formal language. The semantics are presented in the form of conditional equations - equations with conditions attached to them; there may be zero or more conditionals. A condition is an equation in which the *left* and *right hand sides* are compared, and is of the form S = T or S != T for positive and negative tests respectively. However, if one side of a condition is a newly introduced variable, the value of the other side will be assigned to the variable, the value of which can be used in subsequent conditions and the main equation.

Below are three ways in which non-zero conditional equations can be written:

(a) [TagId] S = T when C1, C2, ...

(b) [TagId] C1, C2, ... ====> S = T

(c) [TagId] C1, C2, ...
==========
S = T

where C1, C2, ... are conditions.

1.4 Structure of an ASF+SDF Specification

module M
export – syntax visible to importing modules
sorts S1 S2 S3
— declaration of sorts (non-terminals).
context-free syntax
— declaration of sorts and functions (non-terminals).
— definition of grammar rules in the concrete
— syntax and corresponding to functions in the
— abstract syntax

variables
imports M1 M2 M3 - list of imported modules
hiddens – syntax having *only* local visibility.
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 section.

equations
— conditional equations of the abstract syntax.

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

Table 1 shows the global structure of an ASF+SDF Specification. With this table at hand, a general introduction will be presented in this subsection of global structure of a specification. It is worth pointing out that the structure can be segmented into some five parts beginning at the words, namely, **exports, imports, hiddens, priorities and equations**.

The structure of the segments **export** and **hiddens** are identical and describing one describes the other. The keyword 'export' indicates the part of the module that is visible to all other modules that will be importing the specification. On the contrary, the part of the module that is supposed to have only a local visibility is defined in the section **hiddens**. These segments can further be segmented into the following; **sorts, lexical syntax, context-free syntax, imports, priorities** and **variables**. A **sort** is declared by listing its name in the **sort** section of the module. A sort is basically any non-terminal in the concrete grammar and is defined in the lexical and context-free syntax. The name of a sort should be made up of letters; the first of which must be in upper case. LAYOUT and CHAR are predefined sorts.

In the section of **lexical syntax**, regular grammar and literals can be used to map lexical lexical construction onto a resulting sort. The common additives of regular grammar; repetition, character classes, etc, can be used in a definition. The example below defines the sort 'Sort' as being any number of alpha-numeric characters the first of which must be an uppercase character. An 'Int' is defined as a string of digits containing at least one digit.

```
sorts Sort Int
    lexical syntax
        [A-Z][a-zA-Z0-9]*        -> Sort
        [0-9]+                   -> Int
```

There are well-defined rules to resolve ambiguities at the lexical level.

In the **context-free syntax** section, both the concrete and abstract syntax can be defined. The syntax of this section is very much like the **lexical syntax**, but the construction is more sophisticated; a combination of sorts and lexical tokens can be mapped onto a result sort. Further, there can be other modifications for associativity and the bracket function. Below is an example of a **context-free syntax** section. In this example, multiplication and addition are defined as in-fixed functions taking two integers the result sort being an integer. Further, these two functions have been defined to have a left associativity. An example is given of the bracket function and a function, succ, which combines lexical tokens, 'succ', '(' and ')', and a sort 'Int' to result in the sort 'Int'.

```
    context-free syntax
        Int * Int                -> Int {left}
        Int + Int                -> Int {left}
        "(" Int ")"              -> Int {bracket}
        succ "(" Int ")"         -> Int
```

As mentioned above, variables that will be used in the equations below must be declared in the **variable** section.

A list of modules that are being imported is presented in the section **imports**.

The section of **priorities** is used to define the priority relations between priorities. A priority is defined using the arithmetical comparism operators.

The section of **equations** is where the semantics of the language are associated to the syntax by way of conditional equations. An equation is composed of a tag, a set of conditions and a simple equation.

A typical module of ASF+SDF is contained in two text files - one for the SDF definitions and the other for the ASF. (A real life example is presented in section 2.)

1.5 Rewriting a Specification

When a specification is presented to the ASF+SDF system, it generates a programming environment for it. The environment checks the syntactic validity of the term presented to it and, if there are no errors, reduces the term. The ASF+SDF system uses the *leftmost innermost* reduction strategy; this implies that if a term has sub-terms, the sub-terms will be be evaluated in a left-to-right order before the main term.

Reduction of a term involves parsing the term and matching it against the equations in the specification. A term that matches any equation is called a redex; if the conditions attached an equation are true, the term is rewritten with the rhs of the equation. An evaluated redex is a reduct.

A term matches a rule when for all variables and sub-terms in the *left hand side* of the equation, a binding can be found to instantiate the *left hand side* into the term. Evaluating the rhs involves instantiating variables in rhs with bindings established from evaluating the *left hand side* and the conditions.

It is not defined which equation the system will choose when a term matches more than one equation. This will result in a non-deterministic specification - a term having more than one normal form. This is considered bad coding style and can be prevented by adding distinguishing conditions. Another way to prevent multiple normal forms is that if the tag of an equation contains the word *default*, that equation will be treated as a default and will only be chosen if the term fails to match any other equations except the default equation. A valid term that cannot be matched to any equation is said to be in normal form.

1.6 Debugging a Specification

The Equation Debugger, EDB, provides a debugging facility for the semantics of an equation. Debugging, in its most general form, is the tracing and interruption of the rewrite process. A specification can only be debugged it has passed all syntactic tests. The debugger provides the user with a powerful means to monitor the behavior of a specification.

The current version, which runs concurrently with the evaluating machine, has two modes:- the Step and Go mode. The step mode communicates with the debugger just before an equation is applied to a term and provides detailed information as regards the status of that application. The information includes the success, or failure of applying the term to a chosen equation. The go mode, on the other hand, provides a situation where the rewrite process is only interrupted at the breakpoint, and provides no interim information on the status of the debugger.

The user interface of the debugger can be started by clicking the 'Debugger On' option in the ASF+SDF Meta Environment. The debugger has its own user interface window that is constructed around the term to be evaluated. It has options for adding and deleting patterns, adding and deleting break points and setting of various flags.

It is rather unfortunate that the user cannot alter or affect the rewrite process in any way; he is reduced to an *observer* of the process. It should have been possible to alter the rewrite process considering the debug process run concurrently with the rewrite machine.

1.7 The New Debugger

It has been found desirable for the new debugger to be executed independently of the ASF+SDF system. In this sense, it should be a separate process that makes use of information provided to it by the ASF+SDF system. But before the ASF+SDF system can have any information, it must evaluate the term presented to it and save the information in a file. This explains why the techniques of a *post-mortem* debugger have been chosen

over and above all else. The new debugger should increase flexibility and enhance the user-interface by making use of the current visualization techniques available.

2 Debugging

Decarbonate, defreeze, debug; these words have a touch of negativity about them - remove carbon, freeze or bug respectively. That removing a bug should have anything to do with computer science is remarkable. Legend has it that a bug played a significant part in the failure of an important software project in the days of yore - when computers were more self-respecting and took a whole building to themselves; hence debug.

The essence of debugging is derived from the rarity with which computer programs, initially at least, behave as expected of them by programmers. It is essential to point out that a bug, as referred in this paper, occurs only in syntactically correct computer language constructs. Whereas both syntactic errors and bugs must be corrected from the source code, a source code with a syntactic error will not proceed to the code-generation stage of a compiler. A bug is actually a semantic error and is perfectly acceptable to most compilers.

Dijkstra [DIJK78] and others have proposed that debugging time could be shortened by rigorous reasoning about a program's correctness. In fact, experimental results described in [WEIS82] show that this is exactly what most programmers do and that they only resort to other means when their mental process fails.

2.1 Locating Bugs

The job of a debugger is primarily to locate a bug. Debugging is a difficult job since the programmer has little guidance locating the bugs. As often happened, there is an interval between when the bug occured and the time when the effects of the bug are noticed. This interval only complicates the task of locating the bug. There is the particular difficulty in locating a bug that originates from outside the program; consider faulty computer devices such as the standard output. The location of a bug becomes more vague when programs run in parallel, see [CM85].

2.1.1 BreakPoints

Breakpoints are user-specified conditions that break the execution of a program when the condition is attained. Breakpoints are most useful, when set at vital areas of the program execution in order to speed up the debug process. Two types of breakpoints can be crystalized, namely control breakpoints and data breakpoints. Control breakpoints are break conditions specified in terms of the program's control such as a call to a procedure. Data breakpoints, on the other hand, are conditions defined in terms of the memory state of a program, an example is to set a condition to when a variable has attained a specified value.

In a way, control break points are a subset of data break points. Certainly, control break points are less complex than data break points and the complexity, alone, is an outstanding differentiating factor. Control break points are a one-to-one mapping of a

condition to the fulfilment of the condition. Data breakpoints have a one-to-many relation since there is a multiple of ways in which a memory address can be updated.

Some hardware such as Intel i386 [INT386] and SPARC [SPARC] provide direct means for monitoring memory locations; an essential service in implementing relatively nonexpensive data breakpoints. But there is a catch: the number of memory locations that can be monitored simultaneously is rather very limited. Other ways in which the hardware may provide some inexpensive monitoring include a trap each time the virtual memory page of the location is accessed. This method [VAXDEB], however, is not that satisfactory since a memory location is often a small part of the virtual page and reference to other parts of the page will just as well cause a trap.

Wahbe et al [WLG93] have developed a code patching approach to check memory updates efficiently. Performance gains by [WLG93] has been achieved by using inexpensive techniques to determine whether a target address is part of a data break condition and eliminate update checks on these locations.

2.1.2 Program slicing

Weiser [WEIS82] introduced the method of program slicing as an important step in the debugging process. Program slicing reduces the amount of source code to be inspected when debugging by ignoring the part of the code that is most irrelevant to the bug. A program slice at a point p on a variable v is all statements and predicates of the program that might affect the value of v. Dataflow analysis [ASU88] can be used to automatically find small program slices.

Table 2 shows an example of a program slice. In the example, a slice of the program is shown on variable `total` at line 13. The slice is instructive. Notice how the use of dataflow analysis becomes evident; since `total` depends on variables `x` and `y`, line 4, which modifies the values of `x` and `y`, has been included in the slice.

```
1 program p;                                  program p;
2 var x,y,z,total,sum :  integer;             var x,y,total:  integer;
3 begin                                       begin
4   read(x,y);                                  read(x,y);
5   total := 0;                                 total := 0;
6   sum := 0;
7   if x<= 1 then                               if x<= 1 then
8   sum := y
9   else                                        else
10   read(z);
11   total := x*y                               total := x*y
12   end;                                        end;
13 bf end.                                    end.
```

Table 2 An original program and a slice of it..

2.2 Algorithmic Debugging

The original form of algorithmic debugging, introduced by [SHAP83], is an interactive process where the debugging system aquires knowledge of the expected behaviour of the bugged program. This knowledge is used to locate the bug. However, [SHAP83] has the drawback of being limited to small Prolog programs and an unacceptably large number of interaction with the user. Improvements in the original method by [FSKG92] has broadened the scope of languages to include procedural, fourth generation (4GL) and lazy functional languages. Whereas interactions with the user cannot be eliminated completely, their number have been reduced significantly. Program slicing and algorithmic debugging has been combined to provide improvements in locating bugs, see [SHAH90].

```
1 procedure insert(elem :  integer);
2 var ind :  integer;
3 begin
4   if index=0 then
5     list[1]:=elem
6   else
7   begin
8     ind := 1;
8     while ((ind < MAX) and
9           (list[ind] > elem)) (* bug inserted here.
10                              should be:  elem > list[ind] *)
11     do ind := ind + 1;
12     ShiftRightAndInsert(list,ind,elem)
13   end;
14 end.
```

Table 3 The bugged procedure `insert`..

The example below is the interaction session of an algorithmic debugger. The example involves an erroneous implementation of the insert-sort algorithm. The error, as was found out by this algorithmic debugger, is in the function `"insert"`. (The bug introduced in this example swaps the arguments of the predicate `greater` in `"insert"`.) Sort([2,1,3]) should return [1,2,3].

sort(in:list=[2,1,3], out: sort=[3,1])?
> no
sort(in:list=[1,3], out: sort=[3,1])?
> no
sort(in:list=[3], out: sort=[3])?
> yes
insert(in:elem=1,list=[3], out: insert=[3,1])?
> no
insert(in:elem=1,list=[], out: insert=[1])?
> yes
An error has been localized inside the body of function "insert"

2.3 Comparison of the Debugging of TRS versus conventional languages

This subsection is attributed to Tip[89]. The concepts of Term Rewrite System, TRS, are very different from conventional languages. The notion of procedure, for instance, is unknown in TRS's. However, there are points where these concepts converge. Some features of `breakpoint debugging` mentioned in [FER83] will be compared below for;
1) imperative languages such as PASCAL
2) Prolog
3) a term rewrite system

Size of the smallest execution step

The size of the smallest execution step must be chosen so that the user is not drowned in an unnecessary abundance of information. The smallest step for any computer program is the machine instruction. However, propagating this as the smallest execution step will be quite unacceptable.

The concept of statements is quite inherent to imperative languages and is also the smallest execution step. Statements, however, can be very complicated in C, for instance.

In C-Prolog, a `Procedure Box` is used to express control flow. A procedure box has four ports namely, `Call, Exit, Fail` and `BackTo`. Interested parties may find the functionalities of ports in [PROLOG86]. The smallest step for the C-Prolog debugger is a transition between the ports.

The smallest execution step in a TRS is a rewrite-step. Evaluation of a condition is also a rewrite-step so there must be a way to differentiate such steps from the ordinary rewrite for an equation.

The notion of levels

Imperative languages associate the concept of `level` with the recursion of subprogram calls.

In C-Prolog, the notion is associated with entering and leaving procedures. Backtracking, however, complicates this notion. The user may choose to step through the procedures or skip them all together.

The levels of a TRS are based on the nesting of the evaluation of the conditions attached to the equations.

Breakpoints

Breakpoints in imperative languages are often associated with line numbers in the source code. Control of execution is taken over from the user until the next breakpoint is encountered.

The C-Prolog debugger allows the user to set breakpoints on predicates.

Breakpoints in TRS debuggers can be set to both patterns occuring in the term, or rewrite rules. Setting breakpoints to rewrite rule can be made broader by also having the possibility to set breakpoints on `syntax rules`. Setting a breakpoint on a syntal rule is equivalent to setting a breakpoint on the cluster of rewrite rules generated by that syntax rule.

Inspection of Variables and source-level expressions

Inspection of variables is available for all three types of languages being compared here. It is nearly unthinkable to have a debugger that does not provide this feature. Computation of source-level expressions is a different story all together. Whereas source-level expressions can be computed for nearly all imperative language debuggers and C-Prolog, TRS's may not provide this feature that readily.

Modification of variable values and source code modification

Modification of variable values should, preferably, be available for imperative debuggers. However, program behaviour may not be predictable for variables that determine the control-flow of a program. Modifying the program may at best have no effect on imperative languages, since most imperative language source codes must go through the routine of compilation before being useful. On modification of the source code, some debuggers terminate debugging, compile the source and start debugging anew.

The C-Prolog debugger allows modification of the prolog-database during debugging. Modifying the database, however, can lead to unpredictable behaviour when backtracking re-enters a procedure box whose predicates have been modified.

Modifying the value of TRS variables should be possible but quite often a rewrite rule is chosen solely on the value of a variable. This situation is very much like the situation above for imperative languages where a certain value for a variable determines the control-flow. Changing the value of a variable after a rewrite rule is chosen could lead to an incorrect evaluation of a term. Changes to the source code must be analysed before it is useful so modifying the source code should not be considered.

3 An ASF+SDF Example

The syntax, evaluation and user interface of the Meta Environment of ASF+SDF will be introduced in this section by way of an example.

3.1 The Meta Environment

Figure 1 is the user interface of the Meta Environment, and it has the following buttons:

- Specification
  This button has another sub-menu attached to it; the buttons of the sub-menu are:
  - Add: This button request the name of a module to be added to the specification.
  - Clear: removes all module from a specification
  - Save: writes all edited modules and terms to a physical medium.
  - LateX: choosing this button will result in the production of a latex representation of all the modules in the specification.
  - Debug On: this button is flag that requires whether the debugging facility of the ASF+SDF system should be turned off or on.
  - Quit: obviously quits the Meta Environment.

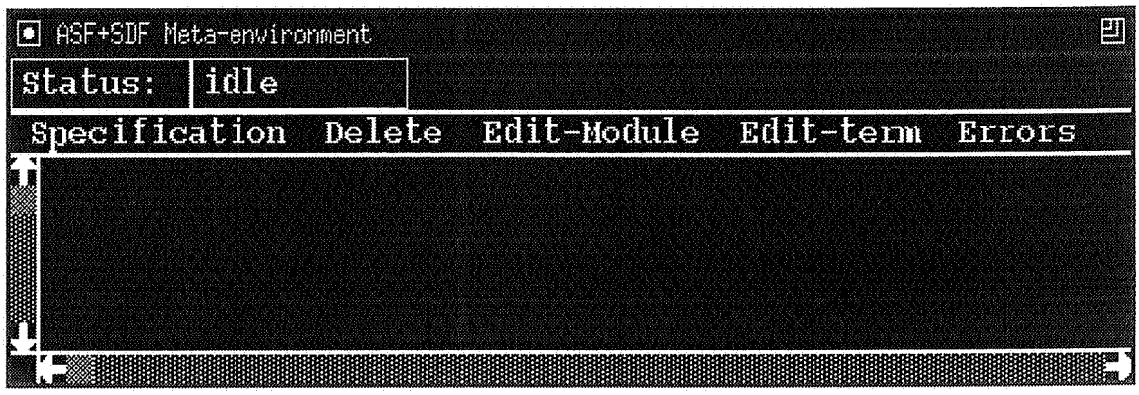


Figure 1: The Meta Environment

- Delete
  When this button is chosen, the user is presented with a list of all the modules in the specification. Clicking the mouse on a particular module, deletes the module from the specification. (Doing this, however, could endanger the integrity of specification if some other modules depend on the deleted module.)
- Edit-Module
  When this button is chosen, the user is presented with a list of all the modules in the specification. Clicking the mouse on a particular module, spawns a Generic Syntax-directed Editor, GSE, to edit the module. The GSE integrates textual and structural editing so that changes in the module are directly reflected in the syntax of a specification.
- Edit-term
  When this button is chosen, the user is presented with a list of all the modules in the specification. Clicking the mouse on a particular module, spawns a Generic Syntax-directed Editor, GSE, to edit a term that can reduced by that module (and all others it imports). This is also the button at which terms can actually be reduced by choosing the *reduce* button. Figure 2 shows a term editor for the example of this section.
- Errors
  This button gives a helping hand on the handling of errors. It has the a sub-menu and below is their functionality.
  - Next requires the system to move on to the next error in the specification.
  - Explain attempt to explain the course of an error, as far as it is explicable.
  - Source give an indication as to the source of an error.
  - Forget request the system to ignore this error. Ignoring an error is not mentioning the same error if it occurs again.

3.2 Booleans - the specification

The example that will be used here is a specification of logical booleans with some of the functions that are applied to booleans.

module Booleans

imports Layout

exports

sorts BOOL

context-free functions

true	→ BOOL
false	→ BOOL
BOOL "|" BOOL	→ BOOL {**left**}
BOOL "&" BOOL	→ BOOL {**left**}
not "(" BOOL ")"	→ BOOL
"(" BOOL ")"	→ BOOL {**bracket**}
xor BOOL BOOL	→ BOOL

variables

Bool [0-9']* → BOOL

priorities

xor BOOL BOOL → BOOL < BOOL "|" BOOL → BOOL < BOOL "&" BOOL → BOOL

equations

[B1] $$\frac{Bool_1 = \text{true}}{Bool_1 \mid Bool = \text{true}}$$

[B2] $$\frac{Bool_1 = \text{false}}{Bool_1 \mid Bool = Bool}$$

[B3] $$\frac{Bool_1 = \text{true}}{Bool_1 \;\&\; Bool = Bool}$$

[B4] $$\frac{Bool_1 = \text{false}}{Bool_1 \;\&\; Bool = \text{false}}$$

[B5] $$\frac{Bool = \text{false}}{\text{not}(Bool) = \text{true}}$$

[B6] $$\frac{Bool = \text{true}}{\text{not}(Bool) = \text{false}}$$

[B7] $$\frac{Bool = Bool_1}{\text{xor } Bool\ Bool_1 = \text{false}}$$

[B8-default] xor *Bool* $Bool_1$ = true

3.3 Booleans - a term

In Figure 2 a valid term of the specification is introduced in the previous section. The term has been partially indented to ease readability and to show the levels of the sub-terms in the term.

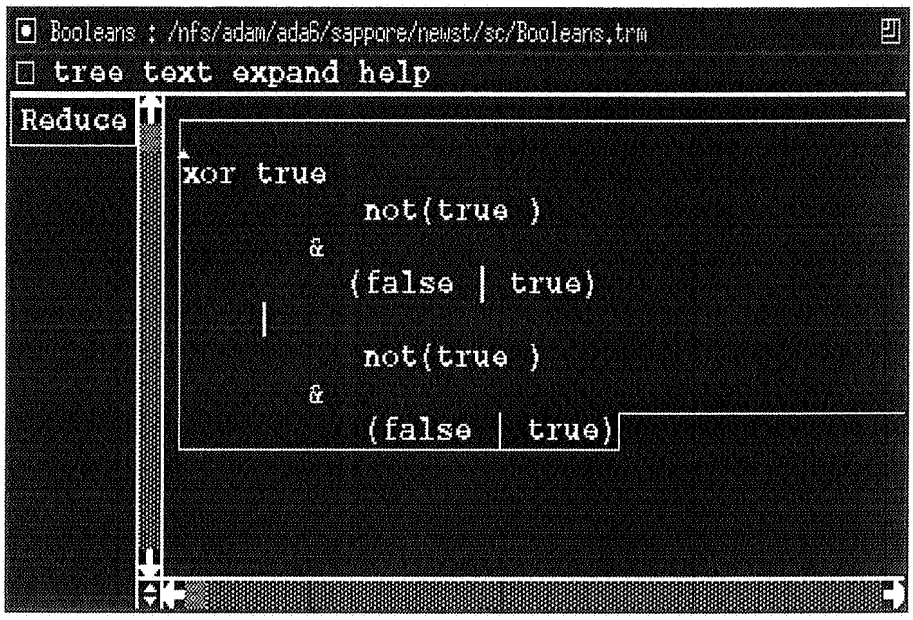


Figure 2: The term

3.4 Booleans - the reduction repertoire

The evaluation process starts by creating a programming environment for the syntax of the *Booleans* specification. The programming environment is in many ways comparable to the generation of a compiler - that subscribes to the grammar of *Booleans*. The term is then parsed according to the grammar of *Booleans* and duly *flattened*. Again, this stage could be compared to compiling a program, in this case a term, by the the *Booleans compiler*. Several error checks - type-check, syntax, etc - will be executed on the term.

Flattening out *the term* will result in a term with the following. sub-terms:

- (a) xor
- (b) true
- (c) *not(true) "&" (false | true) | not(true) "&" false | true*

This is because the function *xor* requires two booleans in order to be grammatically correct. Since the *innermost left-to-right* reduction strategy is being applied, sub-term (b) will be chosen for reduction. However, this is already in normal form so the next leftmost sub-term (c) will be chosen. This is not an atomic term, in that it has other sub-terms, so it will be reduced to a normal form for the reduction to proceed at this point.

There is an ambiguity in term (c) but this is resolved by the fact that priority of " "&"" has been defined to be higher than that of "|". The priority definition, therefore, plays a role in determining how that the term is flattened out into the following sub-terms:

- (c.1) *not(true) "&" (false | true)*
- (c.2) not(true) "&" false | true

Reducing term (c.1) is more interesting; the sub-terms are *not(true)* and *(false | true)*. The first sub-term, *not(true)* can be reduced by applying either equation B5 or B6. The

conditions on both equations, however, narrows down the choice to equation B6 only. The result of reducing the first sub-term of (c.1) is *true*. The ASF+SDF system uses the *lazy - incremental* reduction strategy, and goes off to reduce the *interim* term of c.1' which is *false "&" (false | true)*. The next step will be to reduce c.1' by applying equation B4. The *lazy* qualities of the system will be used in this reduction. It is clear that the value *Bool* in equation B4 is not necessary to determine the result of applying this equation. Accordingly, the second sub-term of term (c.1), *(false | true)*, will *not* be evaluated at all. The reduct of term (c.1) is *false*

Term (c.2) will not be reduced separately - that would have been in contention with the *lazy - incremental* nature of ASF+SDF, rather, the interim term, (c.12), *false | not(true) "&" false | true* comprising the reduct of term (c.1) and term (c.2).

Once again, term (c.12) would have an ambiguous expression had it not been for the left-associativity relationship defined on "|". Therefore, term (c.12) will the following sub-terms:

- (c.12.1) false | not(true) "&" false
- (c.12.2) true

The *lazy* nature will again be exploited in equation B2 to reduce term (c.12.1) to *not(true) "&" false*, which in turn will be reduced by equations B6 and B4 respectively to reduce the term (c.12.1) to its reduct of *false*. The reduct of term (c.12.1) is prefixed to term (c.12.2) and reduced to *true*; which is also the reduct of term (c).

The very last term to be reduced is *xor false true*. There are two equations to choose from. If equation 'B8-default' is chosen the term will always be evaluated to *true*, regardless of the parameters of the function. However, this is not how the function is evaluated. Since the tag of the equation *'B8-default'* includes the string *'default'*, the equation will be treated as default - and will only be chosen when equation B8 fails. Considering the condition of equation B8 which will fail, equation *'B8-default'* will be applied and the term is evaluated to *true*

3.5 Booleans - a summary

A list of the steps taken to reduce the term to a normal form is presented here. The terms are reduced incrementally. The actual sub-term that is reduced is italicized.

- xor false *not(true)* "&" (false | true) | not(true) "&" false | true
- xor false *false "&" (false | true)* | not(true) "&" false | true
- xor false *false | not(true) "&" false* | true
- xor false *not(true) "&" false* | true
- xor false *not(true)* "&" false | true
- xor false *false "&" false* | true
- xor false *false | true*
- *xor false true*
- *true*

4 The Current Debugger

The user interface and functionality of the various buttons in the current debugger will be discussed in this section. Figure 3 shows the user interface of the debugger.

4.1 The User Interface

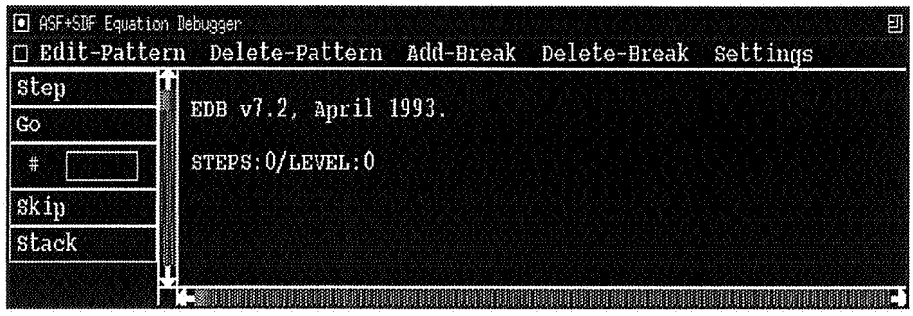


Figure 3: The EDB User Interface

- Step
  When this button is selected, the user is presented with a sub-menu. The sub-menu comprises of an entry field labeled ,'#' Go, Skip and Stack.
  - Step
    To match a term to an equation, the system first considers a list of plausible equations against which the term will be matched. The 'Step' button requires the system to attempt to match the term to one of the equations in the said *list*. Whether or not there is a successful match is irrelevant, a step has been performed if there has been an attempt to match. In effect, the step button reveal a detailed trace into the rewrite procedure explicating why certain equations cannot be executed.
  - Entry field, #
    An integer value can be entered here. This value will be used by the Go button described below. However, if the Return key is pressed during the entry the effect will be equivalent to selecting the 'Go' button.
  - Go
    As mentioned above, the 'Go' button functions in association with the Entry field above. When this button is selected rewriting continues until as many steps as the entry has been rewritten. An entry value of 0 is intended to place the rewrite process at its end. However, the behavior of 'Go' is affected by other conditions. If the debugger is in skip mode, the rewriting of conditions will be ignored. If a break pattern is encountered, the rewrite process will stop. If there are not as many steps in the rewrite process as the entry value, the process will stop at the begin or end of the rewrite process.
  - Skip
    The skip button has the effect of the rewrite process jumping out of a condition

and placing the process at just after the condition. Selecting the button will be ignored if no condition is being rewritten.

  - Stack
    The stack button displays the condition stack, if any, of the rewriting process.

- Edit Pattern
  Editing a pattern involves adding the structure of a term. If a term matches the structure of a pattern, then this will be regarded a break point and rewriting is put on hold.
- Delete Pattern
  This is a corollary of the previous menu; patterns that have been added in that menu may be removed in this.
- Add-Break
  Adding a break point is tantamount to adding the tag of an equation to a list. The rewrite process will be put on hold when a term matches an equation with the same tag.
- Delete-Break
  Again, this menu is the opposite of the previous one; it removes a tag from list of tags that have added at the Add-Break menu.
- Settings
  When this button is selected, the user is presented with a sub-menu. The sub-menu is a check button comprising the following entries:-

  - Show Bindings
    If this flag is set, the bindings will be displayed.
  - Show Equation
    If this flag is set, the equation that matches the term will be displayed.
  - Show Redexes
    If this flag is set, the term or redex that is to be matched against an equation will be displayed.
  - Show Reducts
    If this flag is set the term resulting from having successfully matched the redex to an equation or reduct will be displayed.
  - Instantiate Conditions
    Checking this button will allow the variables, bindings, status, etc. of the conditions of an equation to be displayed.
  - Show Tags
    Checking this button will allow the tag of equations to be displayed.
  - Restrict PP
    Checking this button will allow the pretty printing of all (interim) results.
  - Stop at Breaks
    Checking this button will allow the actually allow the rewrite process to stop at break points that have been previously set.

4.2 Shortcomings

The current debugger runs concurrently with ASF+SDF system; the debugger is actually an integral part of the ASF+SDF system. Unfortunately, this has led to a significant slowdown of the debug process, not to mention the inherent lack of flexibility and modularity. The debugger makes little use of the current visualization techniques available. Some users find it too liberal with the information it provides - they cannot make head or tail from all the information.

The most outstanding shortcoming of the current debugger is that it runs concurrently with ASF+SDF system - this leads a handful of undesirable properties.

It is not possible to fully control the rewriting of conditions - one cannot, for instance, directly determine which conditions of an equation to rewrite and which to ignore. The conditions are treated as though they were one bulk and there is no means to differentiate between them.

4.2.1 Direction of Execution

The rewrite process is, as far as the debugger is concerned, mono-directional; it moves in the *forward* direction. In other words, there is no history of previously executed rewrite steps.

4.2.2 Setting of Break points

A break point can be set only by providing the tag of an equation. Since equation tags are not unique, the effective break point that is set is any equation with that tag. This could be seen as an advantage in its own rights, but when the unicity of a break point is considered this is a drawback.

4.2.3 Spooling of Results

All the results of a rewrite are displayed in one window; resulting in a seemingly overcrowded window. Besides that, the cut-and-paste functions of the X-Window are disabled and there is not that easy to redirect these results to a file.

5 The Metamorphosis

Figure 4 shows the *modus operandi* of the old debugger - a direct co-operation between the debugger and the rewrite engine. The debugger requests the execution of a procedure in the rewrite engine and the rewrite engine, on its part, sends information to the debugger.

Figure 5 is a sketch of the reality of the new debugger; it has no direct link to the rewrite engine. The rewrite engine provides, at a go, all the information the debugger will ever requests in the intermediate file. The debugger refers to the intermediate file for all the information it needs about the rewrite process.

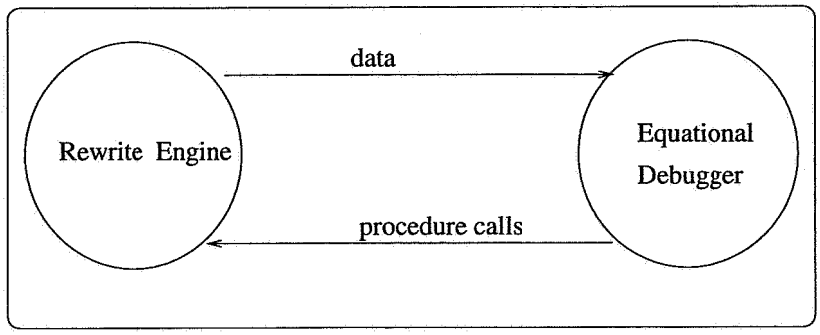


Figure 4: The Structure of the Old Debugger.

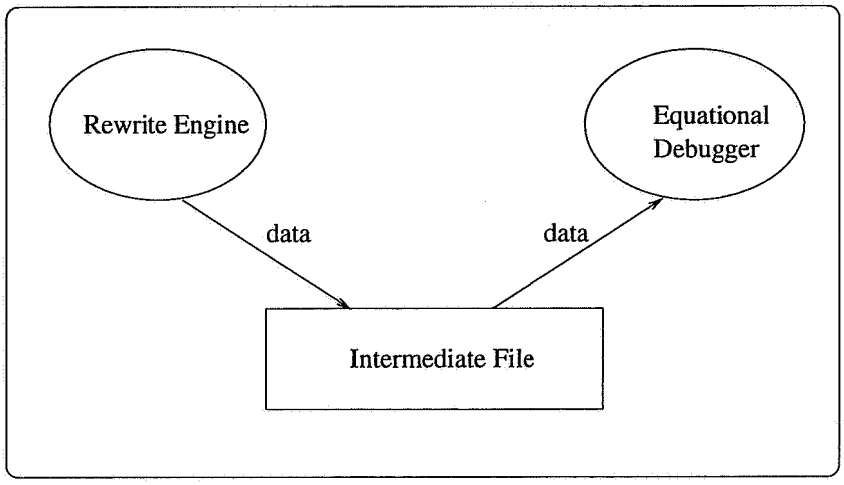


Figure 5: The Structure of the New Debugger.

6 New Features

In this section we shall consider the new debugger; its new features and how it differs from the previous version.

6.1 Handling Conditions

A feature has been added to adequately control the rewriting of conditions attached to an equation. The conditions are not treated as one block; so each condition can be accessed independently of the others. To access a condition, the whole equation will be presented and a condition can be selected by clicking a mouse button. The rewrite process is made to proceed to the first step of a chosen condition. Conditions will be rewritten in the numeric order of their condition number and there is no significant side effect and the access time for one condition is equal to the other.

6.2 Spooling of Results

The results of a rewrite movement are displayed on the screen, a separate window per category; the separate windows are for the display of bindings,equations,redexes,reducts and the current status of the debugger. Besides the display, there is the possibility of writing the same information to a named file. The cut-and-paste functions of the X-Window are enabled on this window.

6.3 Direction of Execution

The rewrite process is, in as far as the debugger is concerned, bi-directional and, moves in both directions : *backward and forward.* In other words, the rewrite process can jump back a number of steps and, seemingly, redo the rewriting.

7 Implementation of the Debugger

The debugger has been implemented as a series of tools that communicate with each other via the ToolBus architecture. Basically, the ToolBus is an architecture that allows a number tools, that are to be combined into a single system, to intercommunicate. Each tool, in turn, has to provide its adapter to the common data representation and message protocols provided by ToolBus. Tools do not communicate directly with each other. A communication repertoire between two tools, T1 and T2, requires T1 to *send* a message to ToolBus which forwards it to T2, T2 must then *receive* the message. ToolBus, in this sense, works very much like the modern mailman. Refer to [BK94] for a detailed description of the ToolBus architecture.

7.1 Overview of tools

- `user-interface` is the tool that handles the interface with the user, it is actually the visible part of the debugger. It creates and destroys all the windows that will be used by the system. Typically, it requests information from the intermediate file and for the textual processing of `ASF+SDF` specifications.

  `user-interface` is a TCL/TK script that will be translated by `wish-adaptor`, a version of TCL/TK.

- `stepper` basically, has the function of reading the intermediate file and storing it as a linked list in computer memory. All other retrievals from the intermediate file are done by this tool. It has a multiple of interfaces with the ToolBus architecture.

  Stepper has been implemented in C++.

- `opener` has the single task of passing the name of the intermediate file to the ToolBus script. It does no other useful task and chances are that with the expected improvements in the command line of the ToolBus utility itself, opener will become obsolete.

  Opener has been implemented in C++.

- `spec2Tblist` has a single interface to the ToolBus. ToolBus provides it with the name of an ASF+SDF equation module. Spec2Tblist reads and analyses the module and returns a TCL/TK list. Appendix B throws some light on the format of the TCL/TK list.

  Spec2Tblist has been implemented in C++.

7.2 The ToolBus script

The ToolBus script that controls the inter-tool communications of the debugger is presented below.

The ToolBus Script

```
process F2LIST(UI:user-interface, SPEC2TBLIST:spec2Tblist) is
let
     FileToOpen : str,
     List       : str,
     Eventnr    : int,
     Steps      : int
in
     rec-event(UI, f2list(Eventnr?,Steps?, FileToOpen?))      .
     snd-ack-event(UI, f2list(Eventnr,Steps, FileToOpen))     .
     snd-eval(SPEC2TBLIST, f2list(FileToOpen))                .
     rec-value(SPEC2TBLIST,f2list(List?))                     .
     if equal(4,Eventnr) then
        snd-do(UI,snd-window(FileToOpen,List))
     else if equal(5,Eventnr) then
        snd-do(UI,spec-list(FileToOpen,List))
     else printf("Bad Event: f2list(%d,%d,%s)",Eventnr,Steps,FileToOpen)
          .  shutdown("Debugger exists without any grace at all")
     fi
     fi
endlet

process KAMIKAZE(
     OPENER     : opener,
     SPEC2TBLIST: spec2Tblist,
     STEPPER    : stepper,
     UI         : user-interface,
     DebugFile  : str) is
let
     Eventnr    : int
in
     rec-event(UI, kamikaze(Eventnr?))        .
```

```
     snd-ack-event(UI,kamikaze(Eventnr))       .
     (
        snd-terminate(OPENER, suicide)        ||
        snd-terminate(SPEC2TBLIST, suicide)   ||
        snd-terminate(STEPPER, kamikaze)
     )                                          .
     snd-terminate(UI, DebugFile)               .
     shutdown("End of Debug Run")
endlet

process STEP-SKIP-ETC (
     STEPPER        : stepper,
     UI             : user-interface
     ) is
 let DelVal         : str,
     Eventnr        : int,
     Steps          : int
 in
     (
        %% Eventnr 1
        rec-event(UI, step(Eventnr?, Steps?, ignorecond))    .
        snd-ack-event(UI,step(Eventnr, Steps, ignorecond)) .
        snd-eval(STEPPER, ignorecond(Steps))
      +
        %% Eventnr 2
        rec-event(UI, step(Eventnr?, Steps?, steps))    .
        snd-ack-event(UI,step(Eventnr, Steps, steps)) .
        snd-eval(STEPPER, steps(Steps))
      +
        %% Eventnr 3
        rec-event(UI, step(Eventnr?, Steps?, skip))    .
        snd-ack-event(UI,step(Eventnr, Steps, skip)) .
        snd-eval(STEPPER, skip)
     ) .
     rec-value(STEPPER,expr(DelVal?))    .
     snd-do(UI, stepvalue(DelVal))
endlet

process DEBUGGER is
 let DebugFile      : str,
     Specifications : str,
     OPENER         : opener,
     SPEC2TBLIST    : spec2Tblist,
     STEPPER        : stepper,
     UI             : user-interface
 in
```

```
  (
    execute(opener,OPENER?)     ||
    execute(spec2Tblist,SPEC2TBLIST?) ||
    execute(stepper,STEPPER?)     ||
    execute(user-interface,UI?)
  ) .
  snd-eval(OPENER, initiate) .
  rec-value(OPENER, expr(DebugFile?)) .
  snd-eval(STEPPER, initiate(DebugFile)) .
  rec-value(STEPPER, spec(Specifications?)) .
  snd-do(STEPPER,readfile) .
  snd-do(UI,name-of-specifications(Specifications)) .
  (
    %% Eventnr 1, 2 and 3
    STEP-SKIP-ETC(STEPPER,UI)
   +
    %% Eventnr 4 and 5
    F2LIST(UI, SPEC2TBLIST)
   +
    %% Eventnr 7=kamikaze=suicide
     KAMIKAZE(OPENER, SPEC2TBLIST, STEPPER, UI,DebugFile)
  )* delta
endlet

tool spec2Tblist is {command = "spec2Tblist"}
tool user-interface is {command = "wish-adapter -script ui.tcl"}
tool opener is {command = "opener"}
tool stepper is {command = "stepper"}
toolbus(DEBUGGER)
```

End of Script

Basically, the script defines a process called *DEBUGGER* and three other auxiliary processes: *STEP-SKIP-ETC, F2LIST* and *KAMIKAZE. DEBUGGER* is the main process and it starts by initiating in parallell the tools *opener, user-interface* and *stepper.* The next step in sequence is that *opener* is requested to provide the name of the debugger's intermediate file; which is passed on to *stepper. Stepper* compiles a list of all the names of the ASF+SDF specifications used for the evaluation of a term and passes this list to toolbus. Thereafter *stepper* goes back and reads the rest of the intermediate file while the said list is passed on to *user-interface.*

After the preparatory steps of the *DEBUGGER*, control is passed on to an alternation of the auxiliary processes: *user-interface* actually takes over the debug process in that the debugger depends on events emanating from *user-interface.*

STEP-SKIP-ETC handles control communication between *stepper* and *user-interface. STEP-SKIP-ETC* succeeds when it receives, and duly acknowledges, an alternation of

appropriate events from, and to, *user-interface*. *STEP-SKIP-ETC* requests a service from *stepper* the result is passed on to *user-interface*.

F2LIST handles control communication between *step2Tblist* and *user-interface*. After receiving the event from *user-interface* . The event contains the name of a file and an event number in the variables *FileToOpen* and *Eventnr* respectively. A request is sent to *SPEC2TBLIST* which returns a list. (The format of the list received from *step2Tblist* is explained in Appendix C.) Depending on the event number received from *user-interface*, an appropriate request is sent to *user-interface*.

KAMIKAZE, receives an event from *user-interface* and promptly acknowledges the event. Thereafter a terminate message is sent to all the tools requesting them to commit suicide.

8 Format of the Intermediate File

8.1 Informal description of the Intermediate File's format

Since information can be of any length or character combination, the notion of escape character has been used to delimit the information and make the underlying structure obvious. Three escape characters: "\,", "\[" and "\]" referred to as Del1, Del2 and Del3 respectively, have been declared for this purpose. There is the requirement, however, that information containing an escape characters combination, should be preceeded by a backslash character.

A *unit* has been defined as the character combination enclosed by Del2 and Del3. A binding has been defined to be two units; one for the name of the variable and the other for its value. A set of bindings delimited with Del2 and Del3 is a Bindingset.

A *Delta* is defined to describe each rewrite step - it includes information such as an equation description, two integers for the step and level numbers, a set of bindings, a reduct and a redex and a set of *deltas* which describe the rewrite steps for the conditions attached to the equation. The intermediate file is ultimately defined as a set containing the name of files and a delta.

The implementation of the intermediate file's structure allows for comments. This is useful for hand-generated intermediate files. A comment is all text that appears just before a delimiter, it is not part of a unit.

module Intermediate File
imports Nint Layout

exports

sorts Del1 Del2 Del3 Unit
Binding Bindingset Reduct Redex FILESET
Delta Eqn_des Log_file String Conditions

lexical syntax

"\\,"	→ Del1
"\\["	→ Del2
"\\]"	→ Del3
"\"" ~[\"\n]* "\""	→ String
Del2 [a-zA-Z0-9_\]\t\n\b\r)(*&\^%$#@!\– <>,./?{}~" :; '"|]* ~[\\] ~[\]] Del3	→ Unit

context-free functions

Del2 String Del3	→ Unit
Unit Unit	→ Binding
Del2 Binding* Del3	→ Bindingset
Del2 {Unit " "}* Del3	→ FILESET
Del2 String String Del3	→ Eqn_des
Unit	→ Reduct
Unit	→ Redex
Del2 Delta* Del3	→ Conditions
Del2 Eqn_des INT " " INT " " Bindingset Reduct Redex Conditions* Del3 Delta*	→ Delta
FILESET Delta	→ Log_file

8.2 Operations on the Intermediate File

Set is by far the most essential operation defined on the intermediate file. *Set* sets a position in the intermediate file where all operations by the debugger are to take place. In a way, the operations are *step, stepfore and stepback, skip and skipback* are all variants of *Set.*

8.3 Example of an Intermediate File

Below is an example of an intermediate file - it is essentially the intermediate file that should be generated for the term shown in Figure 1.

```
\[Booleans\]                \\Name of specifications
\[
    \["Booleans" "B6"\]     \\ Equation applied is has tag B6 in Booleans
    1 0                     \\ step 1 level 0
    \[\[Bool\]\[true\]\]    \\ bindings: Bool = true
    \[false\]               \\ reduct
    \[not ( true )\]        \\ redex
    \[ \]                   \\ no separate evaluation of condition
\]
```

```
\[
    \["Booleans" "B4"\]
    2 0
    \[
        \[Bool\]\[false | true\]
        \[Bool1\]\[false\]
    \]
    \[false\]
    \[false & false | true \]
    \[ \]
\]
\[
    \["Booleans" "B2"\]
    3 0
    \[
        \[Bool\]\[false | not (true ) & false | true\]
        \[Bool1\]\[false\]
    \]
    \[ not (true ) & false | true\]
    \[false | not (true ) & false | true\]
    \[ \]
\]
\[
    \["Booleans" "B6"\]
    4 0
    \[ \[Bool\]\[true\] \]
    \[ false \]
    \[not (true ) \]
    \[ \]
\]
\[
    \["Booleans" "B4"\]
    5 0
    \[
        \[Bool\]\[false | true\]
        \[Bool1\]\[false\]
    \]
    \[ false \]
    \[not (true ) \]
    \[ \]
\]
\[
    \["Booleans" "B8-default"\]
    6 0
    \[
        \[Bool\]\[false\]
```

```
        \[Bool1\]\[true\]
    \]
    \[ true \]
    \[xor true false \]
    \[ \]
\]
```

8.4 Status of the Debugger

This subsection presents the syntax of the operators that maintain the internal status of the debugger. The function *flip* takes a flag and returns its opposite. A *flag* is a boolean value. The functions *init_dbstat* and *init_settin* initializes the status of a file and the *settings* respectively.

module Intermediate Functions

imports Intermediate-File

exports

sorts STATUS-FILE Flag SetVal

lexical syntax

```
"On"                                        -> Flag
"Off"                                       -> Flag
```

context-free function

```
flip "(" Flag ")"                          -> Flag

%% SBind SEq SRedex SReducts SCond STag SSkip
   "{" Flag Flag Flag Flag Flag Flag Flag "}" -> SetVal

   "{" INT INT Flag EQSTACK SetVal "}"      -> STATUS-FILE
   init_dbstat                              -> STATUS-FILE
   init_settin                              -> SetVal

   SetVal "->" "Bind"                       -> Flag
   SetVal "->" "Eq"                         -> Flag
   SetVal "->" "Redx"                       -> Flag
   SetVal "->" "Redu"                       -> Flag
   SetVal "->" "Cond"                       -> Flag
   SetVal "->" "Tag"                        -> Flag
   SetVal "->" "Skip"                       -> Flag

   chSBind     SetVal                       -> SetVal
   chSEq       SetVal                       -> SetVal
   chSRedx     SetVal                       -> SetVal
   chSRedu     SetVal                       -> SetVal
   chSCond     SetVal                       -> SetVal
```

```
  chSTag      SetVal                          -> SetVal
  chSSkip     SetVal                          -> SetVal


variables
  Fl[0-9]*                                    -> Flag
  Flag[0-9]*                                  -> Flag
  SetVal[0-9]*                                -> SetVal
  Sf[0-9]*                                    -> STATUS-FILE
```

8.5 Syntax of Debugger operations

This subsection presents the syntax of the functions that operate on the intermediate file. The functions include *step, stepfore, break, skip, stepback, skipback, breakbc, set* and *setback. Step* take the status of an intermediate file and places the file positions the file pointer to a number of steps forward or backward depending on the a value in the status. *Stepfore* and *Stepback* are variants of *step*, in that they position the said file pointer forward or backward respectively. *Skip* and *Skipback* are responsible for skipping the evaluation of conditions: *skipback*, if it succeeds, *rewinds* the file pointer.

module Debug

imports Status

exports

sorts DB-STATUS Dfile Filepos

context-free functions

```
  ins Eqn_des EQSTACK                         ->  EQSTACK
  "-->"                                       -> Filepos
  FILESET Delta* Filepos Delta*               -> Dfile
  STATUS-FILE Dfile                           -> DB-STATUS
  db "(" FILESET Delta* ")"                   -> DB-STATUS
  step "(" DB-STATUS ")"                      -> DB-STATUS
  stepfore "(" DB-STATUS ")"                  -> DB-STATUS
  break "(" DB-STATUS ")"                     -> DB-STATUS
  skip Dfile                                  -> Dfile
  stepback "(" DB-STATUS ")"                  -> DB-STATUS
  skipback Dfile                              -> Dfile
  breakbc "(" DB-STATUS ")"                   -> DB-STATUS
  set "(" DB-STATUS ")"                       -> DB-STATUS
  setback "(" DB-STATUS ")"                   -> DB-STATUS

  "Pretty-Print" "("  DB-STATUS ")"          -> DB-STATUS

  set_stepcount "(" INT "," DB-STATUS ")" -> DB-STATUS
  set_break "(" Eqn_des "," DB-STATUS ")" -> DB-STATUS
```

```
variables
  Dbfile[0-9]*                          -> Dfile
  Fpos                                  -> Filepos
```

9 Evaluation and Conclusion

9.1 Reading Specifications

Currently, the new debugger has no access to a parse tree of the specifications being debugged. Rather it performs naïve textual searches in the source text.

As it is now, the debugger encounters difficulties analyzing specifications - there is no way to determine and ignore comments, for instance, in the specifications. In ASF+SDF, every single byte specification of a module is inherently part of the syntax or semantics of the specification. The notion of whitespace and comment are foreign. (However, whitespace and comment can be defined as being of the reserved sort LAYOUT; in which case, they will be ignored by the parser of the ASF+SDF system.) Since the debugger deals with the textual level of a specification, it cannot determine what is whitespace and comment - the effect of which is that the debugger completely ignores the syntax part of a specification and views the semantic part in a rather simplistic textual mode.

Below is a segment of a properly defined module. The first line is a comment on the second. Whereas the ASF+SDF system analyses this correctly, the debugger fails in its attempts and may see the first line as a misformed equation.

%% The 'or' operator when the first parameter is true
[B1] $Bool_1$ = *true*
==========
$Bool_1$ | *Bool* = *true*

Another difficulty encountered when dealing with specifications is the identification of the conditions of an equation. The effects of this shortcoming is that the debugger is limited to the representation of conditions of the form below:

[TagId] $Condition_1$,
... ,
... ,
... ,
$Condition_n$
==========
lhs = rhs

This problem could be solved by a tool powerful enough to parse a specification, strip it of all comments and whitespace, and return a version of the specification that is segmented into equations; equations sub-segmented into tag, conditions and *left - and*

right hand side. The debugger as it is now does not have to be changed - the current tool does the same but the list it produces has not been adequately analysed.

9.2 Pattern Recognition

At the textual level, it seems rather impossible to recognize patterns. Consider the module below where *Bool* and *Int* are sorts.

module Sizeof

context-free functions

sizeof "(" BOOL ")" → Int
sizeof "(" Int ")" → Int

For a pattern of the form sizeof(K), it cannot be determined at the textual level whether K is a Bool or an Int. The solution for this problem, is to enhance the information that is received by the debugger. The information must include such detail as the reason to which a certain equation is applied - in fact each equation must be traced to the syntax that justifies it's validity. And if an equation is applied, there should be less difficulty tracing its possible patterns.

9.3 Loop Recognition

Loops, as referred to here, are those that terminate; those that do not terminate cannot be considered since debugging only *starts* after execution has terminated. Some of the advantages that can be accrued if loops could be identified include more flexibility and a significant reduction in the size of the intermediate files that contain a recurring number of similar loops. The user could be informed he is about to enter a loop of a certain size and if he so desires, decide to skip the loop all together.

There is no easy way to recognize these loops but techniques developed for code optimalization in compiler research can be very useful here. The compiler technique of common subexpression could be of particular interest if loop could be viewed as a subexpression. See [ASU92] for more details.

9.4 Influencing the Rewrite Process

It is rather unfortunate that the user cannot alter or affect the rewrite process in any way; he is reduced to an *observer* of the process. This point is intrinsic to the *postmortem* nature of this debugger; ability to alter terms during debugging should be reflected in the term that was rewritten - the debugger only *plays back* the steps done by the rewrite engine.

9.5 Size of the Intermediate File

The size of the intermediate file can be remarkable and it would have been commendable, considering only size, to apply compression methods to it. Another point to consider is the typically short lifespan of an intermediate file; after all, the file holds information on the *term* that has reduced.

References

[ASU88] **Aho, Sethi and Ullman.** *Compilers - Principles, Techniques and Tools.* Adison Wesley, March 1988

[BHK89] **J.A.Bergstra, J.Heering, and P.Klint.** *The algebraic specification formalism ASF. In J.A.Bergstra, J.Heering, and P.Klint, editors, Algebraic Specification, ACM Press Frontier Series, page 1-66.* The ACM press in co-operation with Addison-Wesley, 1989.

[BK94] **J. A. Bergstra and P. Klint.** *The Discrete Time ToolBus* University of Amsterdam Report P9502, March 1995.

[CM88] **B. Miller an J. Choi.** A Mechanism for efficient debugging of parallel programs. *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation.* (Atlanta, 1988), *SIGPLAN Notices,* 23(7) pp. 135-144.

[FSKG92] **P. Fritzson, N. Shahmehri, M. Kamkar and T. Gyimothy** Generalized algorithmic debugging and testing. *ACM Letters on Programming Language and Systems 1,* 4(1992) pp. 303-322.

[HHKR89] **J.Heering, P.R.H. Hendriks, P. Klint, and J. Rekers.** *The Syntax Definition Formalisme SDF - Reference Manual.* SIGPLAN Notices, 24(11):43-75, 1989.

[INT386] **Intel Corporation, Santa Clara, California.** *Intel 80386 Programmer's Reference Manual,* 1986.

[SPARC] **Sparc International.** *The Sparc Architecture Manual.* Prentice-Hall, Inc, version 8, 1992.

[VAXDEB] **B. Beander** Vax DEBUG: an Interactive, Symbolic, Multilingual Debugger. *In Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging ,SIGPLAN Notices,* 18(8), pp. 173-179.

[WEIS82] **M. Weiser.** Programmers use slices when debugging. *Communications of the ACM 25, 7(1982),* pp.446-452.

[WLG93] **R. Wahbe, S. Lucco, S. Graham.** Practical data breakpoints: Design and Implementation. *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation.* (Albuquerque, NM, 1993), *SIGPLAN Notices,* 28(6) pp. 1-12.

A Equation of Intermediate Functions

Below is the equational part of the ASF-SDF specification of the Intermediate functions. The syntax of the specification has been introduced previously in the text.

module Status

equations

[fl1] flip(On) = Off
[fl1] flip(Off) = On

[vl1] $\{Fl\ Fl_1\ Fl_2\ Fl_3\ Fl_4\ Fl_5\ Fl_6\} -> \mathsf{Bind} = Fl$
[vl2] $\{Fl_1\ Fl\ Fl_2\ Fl_3\ Fl_4\ Fl_5\ Fl_6\} -> \mathsf{Eq} = Fl$
[vl3] $\{Fl_1\ Fl_2\ Fl\ Fl_3\ Fl_4\ Fl_5\ Fl_6\} -> \mathsf{Redx} = Fl$
[vl4] $\{Fl_1\ Fl_2\ Fl_3\ Fl\ Fl_4\ Fl_5\ Fl_6\} -> \mathsf{Redu} = Fl$
[vl5] $\{Fl_1\ Fl_2\ Fl_3\ Fl_4\ Fl\ Fl_5\ Fl_6\} -> \mathsf{Cond} = Fl$
[vl6] $\{Fl_1\ Fl_2\ Fl_3\ Fl_4\ Fl_5\ Fl\ Fl_6\} -> \mathsf{Tag} = Fl$
[vl7] $\{Fl_1\ Fl_2\ Fl_3\ Fl_4\ Fl_5\ Fl_6\ Fl\} -> \mathsf{Skip} = Fl$

[ch1] chSBind $\{Fl\ Fl_1\ Fl_2\ Fl_3\ Fl_4\ Fl_5\ Fl_6\} = \{\mathsf{flip}(Fl)\ Fl_1\ Fl_2\ Fl_3\ Fl_4\ Fl_5\ Fl_6\}$
[ch2] chSEq $\{Fl_1\ Fl\ Fl_2\ Fl_3\ Fl_4\ Fl_5\ Fl_6\} = \{Fl_1\ \mathsf{flip}(Fl)\ Fl_2\ Fl_3\ Fl_4\ Fl_5\ Fl_6\}$
[ch3] chSRedx $\{Fl_1\ Fl\ Fl_2\ Fl_3\ Fl_4\ Fl_5\ Fl_6\} = \{Fl_1\ Fl_2\ \mathsf{flip}(Fl)\ Fl_3\ Fl_4\ Fl_5\ Fl_6\}$
[ch4] chSRedu $\{Fl_1\ Fl\ Fl_2\ Fl_3\ Fl_4\ Fl_5\ Fl_6\} = \{Fl_1\ Fl_2\ Fl_3\ \mathsf{flip}(Fl)\ Fl_4\ Fl_5\ Fl_6\}$
[ch5] chSCond $\{Fl_1\ Fl\ Fl_2\ Fl_3\ Fl_4\ Fl_5\ Fl_6\} = \{Fl_1\ Fl_2\ Fl_3\ Fl_4\ \mathsf{flip}(Fl)\ Fl_5\ Fl_6\}$
[ch6] chSTag $\{Fl_1\ Fl\ Fl_2\ Fl_3\ Fl_4\ Fl_5\ Fl_6\} = \{Fl_1\ Fl_2\ Fl_3\ Fl_4\ Fl_5\ \mathsf{flip}(Fl)\ Fl_6\}$
[ch7] chSSkip $\{Fl_1\ Fl\ Fl_2\ Fl_3\ Fl_4\ Fl_5\ Fl_6\} = \{Fl_1\ Fl_2\ Fl_3\ Fl_4\ Fl_5\ Fl_6\ \mathsf{flip}(Fl)\}$

[in0] init_dbstat = {0 0 Off {} init_settin}
[in1] init_settin = {Off Off Off Off Off Off Off}

B Equations of the Debugger functions

This appendix contains the equational part of the ASF+SDF specification of the Debugger functions. The syntax part of the specification has already been introduced elsewhere in the text.

module Debug

equations

[db1] db(*Fset Delta*) = init_dbstat *Fset* $--> Delta$

$$[pp1]\ \frac{Sf = \{int\ int_0\ Fl\ Break\_Pattern\ SetVal\},\ Sf_1 = \{int\ 0\ Off\ Break\_Pattern\ SetVal\}}{\text{Pretty-Print}(Sf\ Dbfile) = Sf_1\ Dbfile}$$

$$[stp1]\ \frac{Sf = \{int\ int_0\ On\ Break\_Pattern\ SetVal\}}{\text{step}(Sf\ Dbfile) = \text{Pretty-Print}(Sf\ Dbfile)}$$

$$[stp2]\ \frac{Sf = \{int\ int_0\ Off\ Break\_Pattern\ SetVal\},\ Dbfile = Fset\ Delta_1\ Fpos\ ,\ int \geq 0 = \text{true}}{\text{step}(Sf\ Dbfile) = \text{Pretty-Print}(Sf\ Dbfile)}$$

$$[stp3]\ \frac{Sf = \{int\ int_0\ Off\ Break\_Pattern\ SetVal\},\ Dbfile = Fset\ Delta_1\ Fpos\ Delt\ Delta_2,\ int > 0 \text{ and } int_0 \geq int = \text{true}}{\text{step}(Sf\ Dbfile) = \text{Pretty-Print}(Sf\ Dbfile)}$$

$$[stp4]\ \frac{Sf = \{int\ int_0\ Off\ Break\_Pattern\ SetVal\},\ Dbfile = Fset\ Delta_1\ Fpos\ Delt\ Delta_2,\ int > 0 \text{ and } int > int_0 = \text{true}}{\text{step}(Sf\ Dbfile) = \text{stepfore}(Sf\ Dbfile)}$$

$$[stp5]\ \frac{Dbfile = Fset\ Delta_1\ Fpos\ ,\ Sf = \{int\ int_0\ Off\ Break\_Pattern\ SetVal\},\ Sf_1 = \{int\ 0\ Off\ Break\_Pattern\ SetVal\},\ int > 0 = \text{true}}{\text{step}(Sf\ Dbfile) = Sf\ Dbfile}$$

$$[stp6]\ \frac{Dbfile = Fset\ Delta_1\ Fpos\ Delt\ Delta_2,\ Sf = \{0\ int_0\ Off\ Break\_Pattern\ SetVal\}}{\text{step}(Sf\ Dbfile) = \text{stepfore}(Sf\ Dbfile)}$$

$$[\text{stp7}]\ \frac{Sf = \{int\ int_0\ \text{Off}\ Break\_Pattern\ SetVal\},\ Dbfile = Fset\ \ Fpos\ Delta_1,\ int < 0 = \text{true}}{\text{step}(Sf\ Dbfile) = Sf\ Dbfile}$$

$$[\text{stp8}]\ \frac{Sf = \{int\ int\ Fl\ Break\_Pattern\ SetVal\},\ Dbfile = Fset\ Delta_1\ Fpos\ Delta_2,\ int < 0 = \text{true}}{\text{step}(Sf\ Dbfile) = \text{Pretty-Print}(Sf\ Dbfile)}$$

$$[\text{stp9}]\ \frac{Dbfile = Fset\ Delta\ Delt\ Fpos\ Delta_1,\ Sf = \{int\ int_0\ \text{Off}\ Break\_Pattern\ SetVal\},\ Sf_1 = \{int\ 0\ \text{Off}\ Break\_Pattern\ SetVal\},\ int < 0\ \text{and}\ int_0 > int = \text{true}}{\text{step}(Sf\ Dbfile) = \text{stepback}(Sf\ Dbfile)}$$

$$[\text{stf1}]\ \frac{Dbfile = Fset\ Delta_1\ Fpos\ Delt\ Delta_2,\ Sf = \{int\ int_0\ Flag\ Break\_Pattern\ SetVal\},\ SetVal -> \text{Skip} = \text{On},\ Dbfile_2 = \text{skip}\ Dbfile}{\text{stepfore}(Sf\ Dbfile) = \text{break}(Sf\ Dbfile_2)}$$

$$[\text{stf2}]\ \frac{Dbfile = Fset\ Delta_1\ Fpos\ Delt\ Delta_2,\ Sf = \{int\ int_0\ Flag\ Break\_Pattern\ SetVal\},\ SetVal -> \text{Skip} = \text{Off}}{\text{stepfore}(Sf\ Dbfile) = \text{break}(Sf\ Dbfile)}$$

$$[\text{stf3}]\ \frac{Dbfile = Fset\ Delta_1\ Fpos}{\text{stepfore}(Sf\ Dbfile) = \text{set}(Sf\ Dbfile)}$$

$$[\text{sk1}]\ \frac{Dbfile = Fset\ Delta_1\ Fpos\ Delt\ Delta_2,\ Delt = Del_2\ Eqn\_des,\ Step,\ 0,\ Bindingset\ Reduct\ Redex\ Cond\_Stack\ Del_3}{\text{skip}\ Dbfile = Dbfile}$$

$$[\text{sk2}]\ \frac{Dbfile = Fset\ Delta_1\ Fpos\ Delt\ Delta_2,\ Delt = Del_2\ Eqn\_des,\ Step,\ Level,\ Bindingset\ Reduct\ Redex\ Cond\_Stack\ Del_3,\ Level \neq 0}{\text{skip}\ Dbfile = \text{skip}\ Fset\ Delta_1\ Delt\ Fpos\ Delta_2}$$

$$[\text{sk3}]\ \text{skip}\ Fset\ Delta_1\ Fpos\ = Fset\ Delta_1\ Fpos$$

$$[\text{br1}]\ \frac{\{Fl_1\ Fl_2\ Fl_3\ Fl_4\ Fl_5\ \text{On}\ Fl\} = SetVal}{\begin{array}{l}\text{break}(\\ \quad \{int\ int_0\ Flag\ \{Eqns\ Eqn\_des\ Eqns_1\}\ SetVal\}\ Fset\ Delta_1\ Fpos\ Del_2\ Eqn\_des,\ Step,\\ \qquad Level,\ Bindingset\ Reduct\ Redex\ Cond\_Stack\ Del_3\ Delta_2) \ =\\ \text{Pretty-Print}(\\ \quad \{int\ int_0\ Flag\ \{Eqns\ Eqn\_des\ Eqns_1\}\ SetVal\}\ Fset\ Delta_1\ Fpos\ Del_2\ Eqn\_des,\ Step,\\ \qquad Level,\ Bindingset\ Reduct\ Redex\ Cond\_Stack\ Del_3\ Delta_2)\end{array}}$$

$$[\text{default-br4}]\ \frac{\begin{array}{c}Dbfile = Fset\ Delta_1\ Fpos\ Delt\ Delta_2,\\ Sf = \{int\ int_0\ Flag\ Break\_Pattern\ SetVal\},\\ Dbfile_1 = Fset\ Delta_1\ Delt\ Fpos\ Delta_2,\\ Sf_1 = \{int\ int_0 + 1\ \text{Off}\ Break\_Pattern\ SetVal\}\end{array}}{\text{break}(Sf\ Dbfile) = \text{step}(Sf_1\ Dbfile_1)}$$

$$[\text{sc1}]\ \frac{\begin{array}{c}Sf = \{int_0\ int_1\ Fl\ Break\_Pattern\ SetVal\},\\ Sf_1 = \{int\ 0\ Fl\ Break\_Pattern\ SetVal\}\end{array}}{\text{set\_stepcount}(int,\ Sf\ Dbfile) = Sf_1\ Dbfile}$$

$$[\text{sc2}]\ \frac{\begin{array}{c}Sf = \{int\ int_0\ Fl\ \{Eqns\}\ SetVal\},\\ Sf_1 = \{int\ int_0\ Fl\ \text{ins}\ Eqn\_des\ \{Eqns\}\ SetVal\}\end{array}}{\text{set\_break}(Eqn\_des,\ Sf\ Dbfile) = Sf_1\ Dbfile}$$

$$[\text{ins1}]\ \text{ins}\ Eqn\_des\ \{Eqns_1\ Eqn\_des\ Eqns\} = \{Eqns_1\ Eqn\_des\ Eqns\}$$

$$[\text{default-ins1}]\ \text{ins}\ Eqn\_des\ \{Eqns\} = \{Eqn\_des\ Eqns\}$$

$$[\text{stb1}]\ \frac{\begin{array}{c}Dbfile = Fset\ Delta_1\ Delt\ Fpos\ Delta_2,\\ Sf = \{int\ int_0\ Flag\ Break\_Pattern\ SetVal\},\\ SetVal -> \text{Skip} = \text{On},\\ Dbfile_2 = \text{skipback}\ Dbfile\end{array}}{\text{stepback}(Sf\ Dbfile) = \text{breakbc}(Sf\ Dbfile_2)}$$

$$[\text{stb2}]\ \frac{\begin{array}{c}Dbfile = Fset\ Delta_1\ Delt\ Fpos\ Delta_2,\\ Sf = \{int\ int_0\ Flag\ Break\_Pattern\ SetVal\},\\ SetVal -> \text{Skip} = \text{Off}\end{array}}{\text{stepback}(Sf\ Dbfile) = \text{breakbc}(Sf\ Dbfile)}$$

$$[\text{stb3}]\ \frac{Dbfile = Fset\ \ Fpos\ Delta_1}{\text{stepback}(Sf\ Dbfile) = \text{Pretty-Print}(Sf\ Dbfile)}$$

$$[\text{skb1}]\ \frac{\begin{array}{c}Dbfile = Fset\ Delta_1\ Delt\ Fpos\ Delta_2,\\ Delt = Del_2\ Eqn\_des,\ Step,\ 0,\ Bindingset\ Reduct\ Redex\ Cond\_Stack\ Del_3,\\ Dbfile_1 = Fset\ Delta_1\ Fpos\ Delt\ Delta_2\end{array}}{\text{skipback}\ Dbfile = Dbfile_1}$$

$$[\text{skb2}]\ \frac{\begin{array}{c} \mathit{Dbfile} = \mathit{Fset}\ \mathit{Delta}_1\ \mathit{Delt}\ \mathit{Fpos}\ \mathit{Delta}_2, \\ \mathit{Delt} = \mathit{Del}_2\ \mathit{Eqn\_des},\ \mathit{Step},\ \mathit{Level},\ \mathit{Bindingset}\ \mathit{Reduct}\ \mathit{Redex}\ \mathit{Cond\_Stack}\ \mathit{Del}_3, \\ \mathit{Level} \neq 0, \\ \mathit{Dbfile}_1 = \mathit{Fset}\ \mathit{Delta}_1\ \mathit{Fpos}\ \mathit{Delt}\ \mathit{Delta}_2 \end{array}}{\text{skipback}\ \mathit{Dbfile} = \text{skipback}\ \mathit{Dbfile}_1}$$

$$[\text{skb3}]\ \text{skipback}\ \mathit{Fset}\ \ \mathit{Fpos}\ \mathit{Delta}_1 = \mathit{Fset}\ \ \mathit{Fpos}\ \mathit{Delta}_1$$

$$[\text{bbr1}]\ \frac{\{\mathit{Fl}_1\ \mathit{Fl}_2\ \mathit{Fl}_3\ \mathit{Fl}_4\ \mathit{Fl}_5\ \text{On}\ \mathit{Fl}\} = \mathit{SetVal}}{\begin{array}{l} \text{breakbc(} \\ \quad \{\mathit{int}\ \mathit{int}_0\ \mathit{Flag}\ \{\mathit{Eqns}\ \mathit{Eqn\_des}\ \mathit{Eqns}_1\}\ \mathit{SetVal}\}\ \mathit{Fset} \\ \quad\ \mathit{Delta}_1 \\ \quad\quad \mathit{Del}_2\ \mathit{Eqn\_des},\ \mathit{Step},\ \mathit{Level},\ \mathit{Bindingset}\ \mathit{Reduct}\ \mathit{Redex}\ \mathit{Cond\_Stack}\ \mathit{Del}_3\ \mathit{Fpos}\ \mathit{Delta}_2) \ = \\ \text{Pretty-Print(} \\ \quad \{\mathit{int}\ \mathit{int}_0\ \mathit{Flag}\ \{\mathit{Eqns}\ \mathit{Eqn\_des}\ \mathit{Eqns}_1\}\ \mathit{SetVal}\}\ \mathit{Fset}\ \mathit{Delta}_1 \\ \quad\ \mathit{Fpos}\ \mathit{Del}_2\ \mathit{Eqn\_des},\ \mathit{Step},\ \mathit{Level},\ \mathit{Bindingset}\ \mathit{Reduct}\ \mathit{Redex}\ \mathit{Cond\_Stack}\ \mathit{Del}_3 \\ \quad\quad \mathit{Delta}_2) \end{array}}$$

$$[\text{bbr2}]\ \frac{\begin{array}{c} \mathit{Dbfile} = \mathit{Fset}\ \mathit{Delta}_1\ \mathit{Delt}\ \mathit{Fpos}\ \mathit{Delta}_2, \\ \mathit{Sf} = \{\mathit{int}\ \mathit{int}_0\ \mathit{Flag}\ \mathit{Break\_Pattern}\ \mathit{SetVal}\}, \\ \mathit{SetVal} = \{\mathit{Fl}_1\ \mathit{Fl}_2\ \mathit{Fl}_3\ \mathit{Fl}_4\ \mathit{Fl}_5\ \text{Off}\ \mathit{Fl}\}, \\ \mathit{Dbfile}_1 = \mathit{Fset}\ \mathit{Delta}_1\ \mathit{Fpos}\ \mathit{Delt}\ \mathit{Delta}_2, \\ \mathit{Sf}_1 = \{\mathit{int}\ \mathit{int}_0 - 1\ \mathit{Flag}\ \mathit{Break\_Pattern}\ \mathit{SetVal}\} \end{array}}{\text{breakbc}(\mathit{Sf}\ \mathit{Dbfile}) = \text{step}(\mathit{Sf}_1\ \mathit{Dbfile}_1)}$$

$$[\text{bbr3}]\ \text{breakbc}(\mathit{Sf}\ \mathit{Fset}\ \ \mathit{Fpos}\ \mathit{Delta}) = \text{step}(\mathit{Sf}\ \mathit{Fset}\ \ \mathit{Fpos}\ \mathit{Delta})$$

$$[\text{default-bbr4}]\ \frac{\begin{array}{c} \mathit{Dbfile} = \mathit{Fset}\ \mathit{Delta}_1\ \mathit{Delt}\ \mathit{Fpos}\ \mathit{Delta}_2, \\ \mathit{Sf} = \{\mathit{int}\ \mathit{int}_0\ \mathit{Flag}\ \mathit{Break\_Pattern}\ \mathit{SetVal}\}, \\ \mathit{Dbfile}_1 = \mathit{Fset}\ \mathit{Delta}_1\ \mathit{Fpos}\ \mathit{Delt}\ \mathit{Delta}_2, \\ \mathit{Sf}_1 = \{\mathit{int}\ \mathit{int}_0 - 1\ \text{Off}\ \mathit{Break\_Pattern}\ \mathit{SetVal}\} \end{array}}{\text{breakbc}(\mathit{Sf}\ \mathit{Dbfile}) = \text{step}(\mathit{Sf}_1\ \mathit{Dbfile}_1)}$$

C Format of List returned by spec2Tblist

The tool spec2Tblist, as mentioned above, processes an ASF+SDF specification and, essentially, returns a list containing information about the specification. The list has information on the number of equations and a description of each equation in the specification. An equation in turn, has the information on the number conditions attached to an equation and the equation itself.

```
The structure of the list built up as follows:
List       ::== <length> <equations>*
length     ::== '<' integer '>'
equations  ::== <length> <conditions> <equation>
line       ::== <length> <string>
conditions ::== <length> <lines>*
equation   ::== <lines>
```

A <string> is defined as a sequence of characters of a fixed length whereas an <integer> is of the traditional defition.

D Example of a debug trail

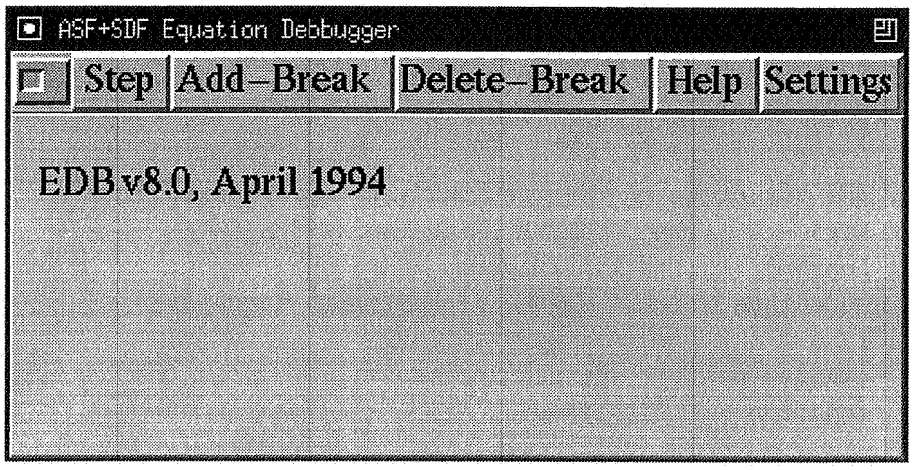


Figure 6: The Debugger - the user interface

In this section the term shown in Figure 1 will be debugged. The intermediate file generated for this term is shown in the previous section. We will begin by introducing the user interface of the debugger and continue to show how this is applied to the various instances of the term from its initial term right through the *normal form*.

Figure 6 shows the basic user interface for the new debugger - a climb down from the old. The buttons that can be chosen from this interface, shown on the first row, are Step, Add Break, Delete Break and Settings. On selecting the Step button one gets a sub-menu as shown in Figure 7.

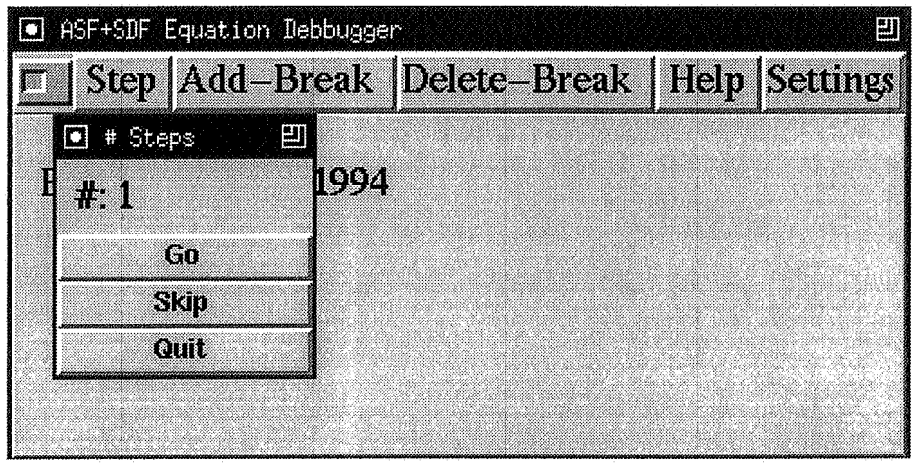


Figure 7: The Debugger - Step chosen

The menu button "Add Break" is chosen to set a break point. Setting a break point is a two-step procedure: it first requires the user to choose a module from which the break point is to originate and then the equations of the chosen module are presented

in a window. The first step, however, is necessary only when there is more than one specification. In this example, there is only one module so the first step is bypassed. A break point is selected by clicking button three on the chosen equation. Figures 8 and 10 show how, for this example, a break point was set on the last equation of the module. The chosen equation will be highlighted and added to a list of break points. (Figure 8 is not a reality and has been added only to show how it would have been had there been more than one module in the specification.)

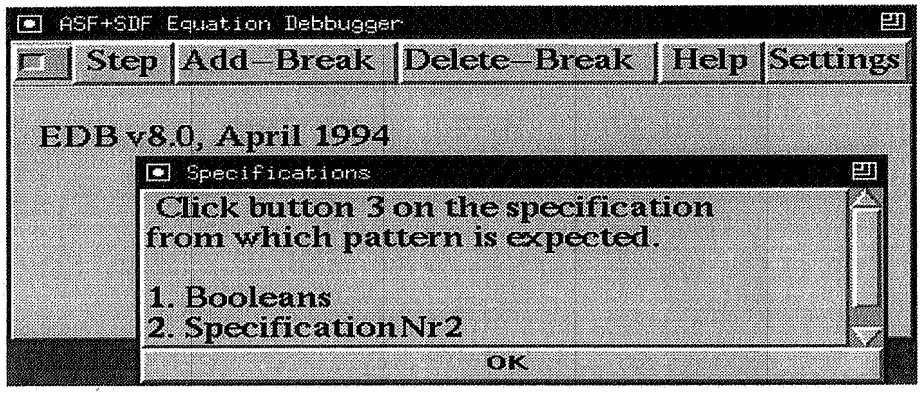


Figure 8: The Debugger - choosing a specification

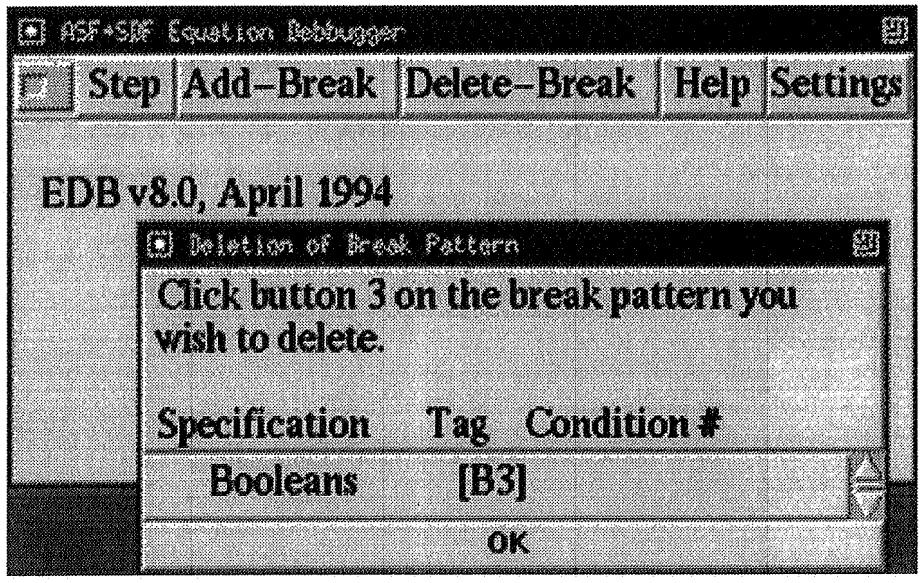


Figure 9: The Debugger.5

Figure 12 show a list of windows - the result of having done a trace of 1 step; the windows in *Bindings, Equation and 'Conditional jump to [B6]'*. The windows *Bindings, Equation* are shown because they were requested *Settings* to *show bindings* and *equation*. The other window, *'Conditional jump to [B6]'* is particularly interesting - it has been shown because the equation has a condition and it provides a way to influence the rewriting the condition.

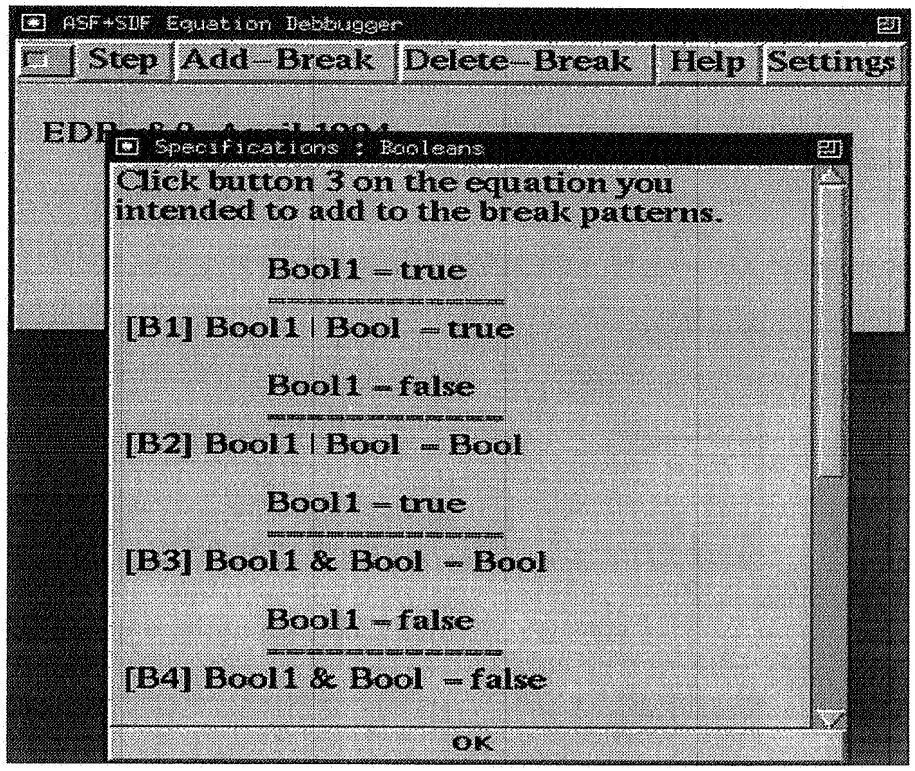


Figure 10: The Debugger - Setting a break point

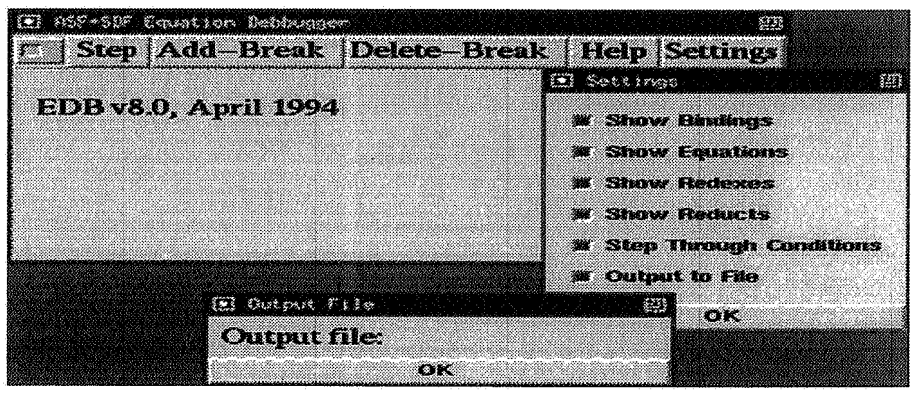


Figure 11: The Debugger.6

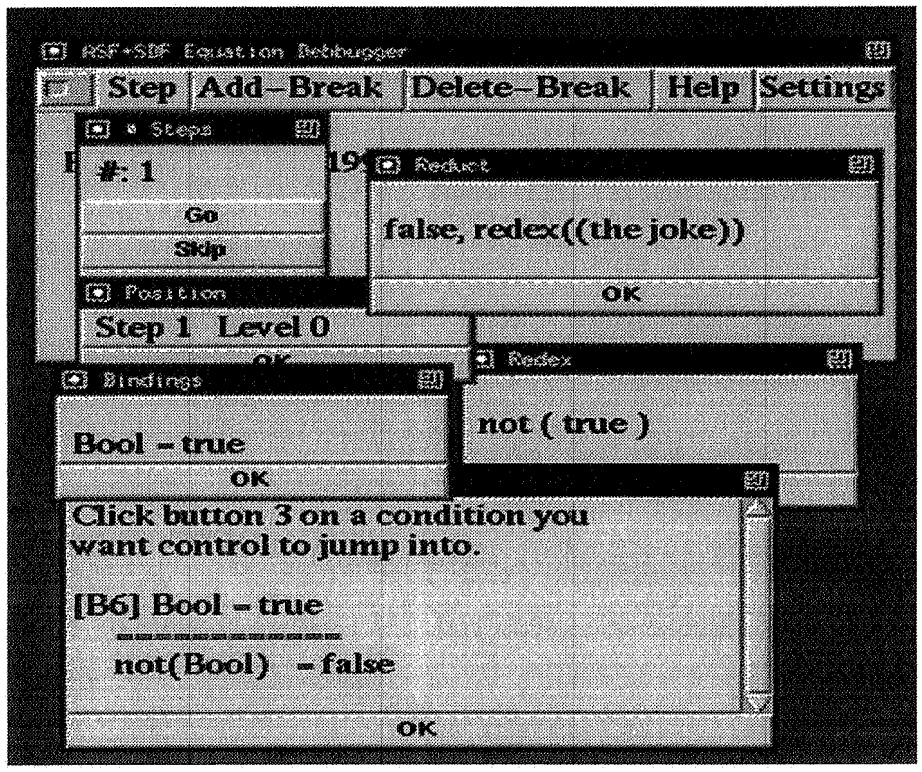


Figure 12: The Debugger - A step taken