ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

M P A Sellink (Ed)

2nd International Workshop on the Theory and Practice of Algebraic Specifications, Amsterdam 1997

Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications, Amsterdam, 25-26 September 1997

Implementation of a prototype for the new ASF+SDF Meta-Environment

M.G.J. van den Brand, T. Kuipers, L. Moonen and P. Olivier

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

ISBN: 3-540-76228-0

Implementation of a Prototype for the New ASF+SDF Meta-Environment

Mark van den Brand

Programming Research Group, University of Amsterdam
Kruislaan 403, NL-1098 SJ Amsterdam, The Netherlands

Tobias Kuipers

Department of Software Engineering, CWI
P.O. Box 94079, NL-1090 GB Amsterdam, The Netherlands

Leon Moonen

Programming Research Group, University of Amsterdam
Kruislaan 403, NL-1098 SJ Amsterdam, The Netherlands

Pieter Olivier

Programming Research Group, University of Amsterdam
Kruislaan 403, NL-1098 SJ Amsterdam, The Netherlands

Abstract

The ASF+SDF Meta-Environment has become a legacy system over the last few years. This paper describes the first steps towards a new implementation of this system. This implementation is based on the latest techniques concerning the coupling of software components, construction of user interfaces and modern programming languages. Special care has been taken to ensure the flexibility and extensibility of the system, both now and in the future.

The general architecture of the new environment is discussed as well as the components which are currently implemented and operational in the environment. Each component is independent of the other components and communicates using the TOOLBUS.

1 Introduction

In the beginning of the eighties the design and implementation of the current version of the ASF+SDF Meta-Environment [21] was started. On top of CENTAUR [6] a programming environment (generator) for writing language definitions in ASF+SDF [16, 3, 12] was developed. An overview of these activities can be found in [17].

The implementation could be considered a test case for all kinds of ideas concerning the lazy and incremental generation of scanners [19], parsers [18], and term rewriting machines. The development of advanced hybrid editing techniques [22], origin tracking techniques [11], incremental rewriting [26], automatic generation of unparsers [10], debugging facilities of term rewriting [30], and the generation of $\text{\LaTeX}$ code [31] were performed in or with this implementation as well.

The current implementation of the ASF+SDF Meta-Environment has a number of drawbacks and shortcomings, the most important ones are listed in below:

- The user interface is old-fashioned and badly organised: reduced terms are not shown to the user via term-editors and long flat lists of modules for deleting and editing of modules and terms.
- An often heard complaint was: “The editor is too restricted, why is it not emacs- or vi-like?”

Figure 1: General architecture of new ASF+SDF Meta-Environment.

- The creation of stand-alone environments is not possible.
- It is impossible to port to different architectures. Limited availability of LELISP [20] on various platforms. Our version of LELISP is becoming obsolete.
- The tree formalism VTP [1] is not easy to use, the connection LELISP/VTP is complicated.
- New research ideas are hard to implement.
- The current monolithic system is hard to maintain. Bugs are not fixed anymore, because the knowledge about the intrinsics of the system needed to fix these bugs is no longer present.

These points show that the system has all the signs of a *legacy system*, mainly because most of the coding has been done by Ph.D. researchers, and consequently the project has had a large turnover of staff. More detailed lists of complaints and shortcomings together with the requirements for the new ASF+SDF Meta-Environment can be found in [7].

These complaints initiated a redesign and re-implementation of the ASF+SDF Meta-Environment. Initially, it was believed that an incremental re-implementation of the ASF+SDF Meta-Environment was feasible, and therefore a number of people started working on the design and implementation of a new user interface and the replacement of the text editing facilities of GSE [22] by Emacs and Epoch in 1992 [23]. However, it proved that it was impossible to manage the interaction between the different tools. This initiated the development of the TOOLBUS, a software interconnection architecture [5, 4] which takes care of the communication of software components. This TOOLBUS will be the backbone of the implementation of the new ASF+SDF Meta-Environment.

Based on the experiences gained with the Epoch-GSE-UI coupling the decision was made to design and implement the new ASF+SDF Meta-Environment from scratch. The fact that the version of LELISP on which the ASF+SDF Meta-Environment is based may become obsolete in the near future makes a “from scratch” approach even more urgent. In this paper we discuss the first prototype of the new ASF+SDF Meta-Environment based on the TOOLBUS. This prototype has very restricted functionality but offers an extendible infrastructure to experiment with various designs.

In the rest of this paper the most important components of the new ASF+SDF Meta-Environment are presented. In Section 2 the architecture of the new ASF+SDF Meta-Environment is discussed. Section 3 describes various repository to store information on ASF+SDF modules, furthermore the tree representation format is briefly discussed. The user interface is discussed in Section 4, the structure editor in Section 5, and the interpreter in Section 6.

2 General Architecture

The architecture of the new ASF+SDF Meta-Environment is depicted in Figure 1. This figure is a snapshot of the current state of the prototype, see Table 1 for a more detailed list of integrated components and their implementation languages. The future architecture of the new ASF+SDF Meta-Environment will contain more components, such as, parsers, unparsers, debuggers, a parser generator, an unparser generator, and an evaluator generator.

Table 1 gives an overview of all currently available components in the prototype. For each component it is listed whether this component is specified and in which language it is implemented.

2.1 ToolBus

Our whole design is based on the TOOLBUS software coordination architecture [4, 5] which utilizes a scripting language based on process algebra [2] to describe the communication between software tools. A TOOLBUS *script* describes a number of processes that can communicate with each other and with *tools* existing outside the TOOLBUS (Figure 2). A tool is more or less equivalent to an operating system process. A language-dependent adapter that translates between the internal TOOLBUS data format and the data format used by the individual tools makes it possible to write every tool in the language best suitable for the task(s) it has to perform.

Figure 2: The TOOLBUS software application architecture

Processes are described as expressions which are built using the TOOLBUS primitives and process composition operators. The following subsections give an overview of the most important TOOLBUS primitives and operators. For a more complete description of TOOLBUS expressions and primitives see [5].

2.1.1 Communication Inside the TOOLBUS

There are two mechanisms available for processes in the TOOLBUS to communicate with each other, message passing and selective broadcasting. A process can synchronously send a *message* using the `snd-msg` primitive which must be received by another process using the `rec-msg` primitive. A process can send a *note* using `snd-note` to all processes that have subscribed, using `subscribe`, to that particular note type. The receiving processes read notes asynchronously using `rec-note`, at low priority. Transmitting notes amounts to *asynchronous selective broadcasting*.

Activity	Specification Language	Implementation Language
ATerms	ASF+SDF	C, Java
AsFix	ASF+SDF	C
TOOLBUS	ASF+SDF	C
Structure editor	ASF+SDF	Java
Text editor		Tcl/Tk
Interpreter	ASF+SDF	C
Tree repository	ASF+SDF	C
Import manager	ASF+SDF	C
User interface		Tcl/Tk, TclDot

Table 1: Components of the new ASF+SDF Meta-Environment which are finished.

2.1.2 Communication Between TOOLBUS and Tools

A TOOLBUS process can initiate communication with a tool by sending a message to a tool using `snd-do` when no answer is expected or `snd-eval` when an answer is expected. A process can receive the answer to a `snd-eval` request using the `rec-value` action.

A tool can initiate communication by sending an *event* to the TOOLBUS. A TOOLBUS process receives this event using the `rec-event` primitive and must acknowledge the event using the `snd-ack-event` primitive.

Furthermore, the execution and termination of the tools attached to the TOOLBUS as well as their connection/disconnection can be controlled explicitly by appropriate primitives.

2.1.3 Process Composition

More complex processes can be created using process composition operators for *choice* (+ operator), *sequential composition* (· operator), *parallel composition* (| | operator), *iteration* (* operator) and *guarded (conditional) execution* (the `if-then-fi` operator). The *process creation* primitive `create` can be used to create new processes.

2.1.4 Types and Variables

All terms within the TOOLBUS are *typed*. The TOOLBUS defines a number of basic types for booleans, integers, reals, strings, and binary strings. Complex types can be formed using a list constructor or function application. The type term is a supertype of all other types. The `let-in-endlet` construction makes it possible to declare variables.

2.2 Implementation

The new ASF+SDF Meta-Environment is based on the TOOLBUS. This means that the TOOLBUS is used as a communication mechanism for the various components available in the environment. The components of the new ASF+SDF Meta-Environment can not communicate directly with each other. The TOOLBUS script takes care of *all* communication between the components. This script should allow a maximal freedom in order to facilitate future experiments, for instance addition of new components.

The information exchange between components is done using the ATerm format, see Section 3.1. The TOOLBUS already provides a debugging facility for TOOLBUS scripts. Currently, we are working on the implementation of a sophisticated debugging technique which allows the debugging of components as well. We plan to extend this technique to support the debugging of specifications as well.

2.3 Discussion

First of all, the parser and parser generator [33] have to be integrated in the new ASF+SDF Meta-Environment. Second, the unparser and unparser generator [10] have to be integrated as well as the $\text{ToL}^{\text{A}}\text{T}_{\text{E}}\text{X}$ facilities [31]. This should be done on top of the BOX language. Third, the ASF+SDF Meta-Environment should provide facilities to compile specifications in order to generate stand-alone environments.

3 Repositories

There are several repositories in the current prototype of the ASF+SDF Meta-Environment. The (abstract) syntax trees in AsFix-format are stored in a tree-repository. The import relations are stored in a separate repository. Finally, the interpreter has its own repository to store the equations which are used to rewrite terms. All repositories were first specified in ASF+SDF but for efficiency reasons implemented in C. Before discussing the implementation details of the repositories, we discuss the general format of all stored and exchanged information.

3.1 Tree Representation

In the old ASF+SDF Meta-Environment the abstract syntax trees are represented by means of VTP (Virtual Tree Processing formalism) [1] offered by CENTAUR [6]. There are two problems connected to VTP: it is hard to learn programming in VTP, furthermore VTP does not offer enough facilities to prevent illegal access to constructed trees. The latter drawback causes a number of the maintenance problems in the old ASF+SDF Meta-Environment.

These “VTP-problems” led to the development of an alternative formalism to represent syntax trees called AsFix. The AsFix formalism is an instantiation of a generic *annotated term* format: ATerms [8].

ATerms are used to represent structured information to be exchanged between a heterogeneous collection of tools. The ATerms format should be independent of any specific tool, but it should be capable of representing *all* data that is exchanged between tools. Consider the following example ATerms:

```
constant    abc
numeral     123
literal     "abc" "123"
list        [] [1, "abc", 3] [1, 2, [3,4], 5]
functions   f("abc")
annotation  f(123){color("red"),path([0,2,1])}
```

The data format used in the TOOLBUS is also based on ATerms. So all functions for processing, constructing, and accessing ATerms can be used on the TOOLBUS level as well.

These functions are formally specified in ASF+SDF [8] and this formal specification is used to make a library of C functions to manipulate terms.

3.1.1 AsFix

ATerms are sufficient to encode parse trees (including optional annotations) for programs or specifications in any language. AsFix is an instance of ATerms used for representing the *parse trees* of ASF+SDF specifications. Parse trees mean that all keywords, whitespace, comments, etc. are preserved in the tree representation. A self descriptive representation is used, i.e., each application of a context-free syntax rule contains a copy of that rule. Consider the following set of function symbols:

- $\text{prod}(L)$ represents the production rule L .
- $\text{appl}(T_1, T_2)$ represents the application of production rule T_1 to the arguments T_2 .
- $\text{l}(L)$ represents literal L .
- $\text{w}(L)$ represents whitespace L .

With these function symbols we can use ATerms to represent parse trees. Consider the following context-free syntax rules of the language defining Boolean expressions.

```
sort Bool
context-free syntax
  true      -> Bool
  false     -> Bool
  Bool and Bool -> Bool {left}
```

The sentence “true” can be represented by the following parse tree:

```
appl(prod("true -> Bool"), [l("true")])
```

The parse tree for the sentence “true and true” is:

```

appl(prod("Bool and Bool -> Bool"),
  [appl(prod("true -> Bool"), [l("true")]),
    w(" "), l("and"), w(" "),
    appl(prod("true -> Bool"), [l("true")])
  ])

```

Not that this parse tree is completely self-contained and does not depend on a separate grammar definition. The “prod”s are extremely simplified. The AsFix representation of parse trees contains a lot of redundant information, such as the “prod”s. Because of an efficient term sharing mechanism, provided by the C implementation of the ATerms library, this causes no problems. The term sharing mechanism takes care of all duplicate terms in the AsFix representation of a parse tree by replacing these terms by a single address representing this term.

3.2 Implementation

The “tree-repository” contains the AsFix representation of all modules of a specification under construction and of all terms being edited or rewritten. The tree-repository provides functions to add or remove a module or term, and to clear the entire repository. It is possible to check whether a module or term is already in the repository. Furthermore, given the name of a module it is possible to retrieve a specific section of a module, its import section, its equations, etc. The tree-repository is implemented as a table with the module name and term name as key and the AsFix representation as value.

The “import-repository” provides all information concerning the import relations of an ASF+SDF specification. It contains the import sections of all modules of a specification in the tree-repository. The import-repository provides functions to add the import section of a new module, to remove the import section of a given module, and to clear the entire repository. Of course, it is possible to retrieve the imports, given a module name. It is also possible to retrieve which modules are not yet in “store” given a list of module names. Finally, a number of operations are provided to calculate and retrieve the transitive closure of the import relation for a given module.

The “equations-repository” of the interpreter will be discussed in Section 6. In fact, there are *no* facilities to retrieve any information from this repository by other tools than the interpreter itself.

3.3 Discussion

The current implementation of the import-repository and the equations-repository are based on a lazy mechanism. Only when import relations or equations of some module are needed is the information derived. It is then stored to be used later on. If the contents of the tree repository are changed both the import and equations repositories are cleared. There is no smart incremental updating algorithm implemented yet.

The information stored in the tree-repository could be extended with all kinds of extra information such as size of the file, creation date, etc. Furthermore, the tree-repository should provide a sophisticated querying mechanism as described in [9]. Such a mechanism can be used to locate definitions of sorts, lexical and context-free syntax rules.

4 User Interface

The user interface of the new ASF+SDF Meta-Environment is built around a visual representation of the import graph of the ASF+SDF specification which is loaded. The major advantage of having such a visual representation as a basis for the user interface is the increased insight in the structure of the specification. Furthermore, effective visualisation of this graph can reveal interesting characteristics of the specification (e.g. repeating patterns and unintended transitive import relations).

The user can select one or more modules in this graph and perform actions on them. Currently, the following actions are supported:

- open the module editor for this module,
- open a term editor over this module,

Figure 3: Browsing the import graph in the new ASF+SDF Meta-Environment.

- delete the module from the repositories,
- request additional information about the module, and
- revert the module in the repositories to the last version saved to disk.

These actions can be selected in a menu which pops up when the user clicks on a module in the graph, or using the module list and buttons on the right side of the screen. In addition to the actions described above, the user can:

- add a module to the specification loaded in the repositories,
- clear all repositories,
- revert all modules loaded in the repositories to the last versions saved to disk, and
- save the import graph in various graphics formats.

Figure 3 contains a screendump of the user interface of the current prototype. From left to right one can clearly distinguish the visual representation of the import graph, the list of modules loaded and the buttons to perform actions on modules.

4.1 Implementation

The user interface is implemented in Tcl/Tk [29] and TclDot: an extension for Tcl that incorporates the directed graph facilities of *dot* [24] into Tcl and provides a set of commands to control those facilities. Basically, TclDot contains

commands to define a graph, add nodes and edges to this graph, and compute the placement of nodes and edges of this graph. The layout is computed in a way that tries to expose the logical structure of the graph, avoid crossings of edges, keep the edges short and emphasize symmetry and balance [15]. TcIDot is part of the *graphviz* package [28, 13]: a set of graph drawing tools for Unix or MS-Windows from AT&T/Lucent Technologies.

As seen from a TOOLBUS perspective, the user interface primarily consists of two methods that accept a message: one to denote addition of a module to the repository and one to denote the import relation between modules. These methods use TcIDot to define the graph and compute its layout. Other methods exist to set the status message and to display additional information about modules.

The user interface sends events to the TOOLBUS to add, delete, revert, or edit a module, to edit a term over a module, to display extra information about a module, or to quit the ASF+SDF Meta-Environment. These events are handled by the TOOLBUS-script which may distribute them to other tools for further processing.

4.2 Future Work

Future work on the user interface includes a mechanism for searching in modules (in cooperation with the tree repository and the editor). Using this mechanism it should for example be possible to highlight/select all modules that use a given function or sort.

Furthermore, the current version of TcIDot only supports static graphs. This means that the layout of a graph is computed from scratch every time an update is performed (i.e. adding or removing a node or edge). The result of this computation can be completely different from the original graph. This is not desirable for a user interface since it can be confusing and the user needs to familiarise himself with a new structure. A new version of TcIDot, called TcIDG, supports so called *dynamic graphs* [27, 14]. The layout of these graphs changes incrementally when updates are performed. This results in more gradual changes in the structure of the graph. As soon as the TcIDG package is released, it should be incorporated in the user interface.

Finally, it is convenient if the user can adapt the layout of the import graph to clarify its logical structure (eg., move nodes/edges to improve their ordering). These edit operations should cooperate in some way with the layout mechanism so that changes of the user are not undone by layouting the graph. An example of a more rigorous editing operation is the ability to define subgraphs in the import graph which can be collapsed into a single node. Such a feature can be useful to improve the readability of big import graphs.

5 Editors

The structure editing system in the new implementation provides roughly the same functionality as the Generic Structure Editor (GSE [22]). There are, however a number of differences, both in the external and the internal behaviour.

5.1 Internal Behaviour

The structure editing system consists of two parts. One is a *structure* editor, the other is a *text* editor. The structure editor operates on parse trees (encoded as AsFix terms). It only manipulates (sub)trees, i.e., it does not manipulate the lexical content of nodes in a parse tree. The text editor operates on a character level, it *does* manipulate the content of nodes in a parse tree.

Both the text editor and the structure editor have a well defined external behaviour (a ToolBus interface) [25]. This makes it possible to use any (existing) text editor as long as it adheres to the interface. One of the main weaknesses of GSE has always been its limited text editing capabilities. By separating text and structure editing functionality we hope to address this problem.

The text and structure editors are tied together by means of a ToolBus script. The script provides us with a one-to-one mapping from text to structure and back. It makes sure that at any given time the data structure in the text editor (the text) can be translated to the data structure in the structure editor (the parse tree), and vice versa. If a string α is a syntactically correct string in the language L , and we have a parser Π_L over this language and a pretty printer pp then $pp(\Pi_L(\alpha)) = \alpha$. As a consequence, if t is the parse tree that results from $\Pi_L(\alpha)$ then also $\Pi_L(pp(t)) = t$.

5.2 External Behaviour

If the edited text is not syntactically correct (which is inevitable during editing) then the smallest subtree that contains an incorrect program fragment will be held in a focus. In GSE a similar approach is used. The main difference between GSE and the new structure editor is that GSE has two specific modes, one for text editing and one for structure editing. Once the switch to text-editing mode is made, all structure information is lost.

The new editor does not need this distinction. It allows text-editing while retaining structure information. Furthermore, the new structure editor can create multiple focuses, thus minimizing the amount of text that needs to be (re)parsed after an edit session.

The difference between these approaches is perhaps best illustrated with an example.

Consider the following program in the language While (See [32] for a specification).

```
x := 10 ;
while x > 0 do x := x - 1
```

Now suppose we want to decrease x with 2 during each iteration. We replace the character 1 with the character 2. After this replacement the focus will be on the integer 2 (The underlined character).

```
x := 10 ;
while x > 0 do x := x - 2
```

Now suppose we want to edit the stop condition of the while loop, such that the loop terminates when x is greater than 2. In GSE the focus would then look like this:

```
x := 10 ;
while x > 2 do x := x - 2
```

In the new editor, instead of increasing the focus, a new focus will be created, which looks like this:

```
x := 10 ;
while x > 2 do x := x - 2
```

One of the motivations for using structure editors is the fact that they allow us to parse text incrementally. Only the parts of the text that have been changed need to be reparsed. As shown in the last example: this strategy obviously results in less parsing than in GSE.

This is not always the case. If we take the original program again, and decide to put the body of the while loop between brackets, we get the following focus positions.

```
x := 10 ;
while x > 0 do {x := x - 1}
```

where in GSE we would have had

```
x := 10 ;
while x > 0 do {x := x - 1}
```

In this case, the last solution is better, because the first solution leaves us with two syntactically incorrect focuses. However, there are a number of strategies that could help us here. In this case, the solution would be to create a new focus that exactly contains all the old focuses, effectively giving the same functionality as GSE.

5.3 Implementation

As stated above, the editing system has been implemented as two separate tools. The structure editor was specified in ASF+SDF. This specification can be used as implementation using the interpreter described in Section 6. Because of the experimental nature of the system in this stage, it proves to be difficult to use the interpreter at two levels at the same time (as a rewrite machine for specifications written using the ASF+SDF Meta-Environment, and as a implementation of the editor this specification is written in). Besides, the interpreter is not (currently) fast enough to support interactive use.

We therefore implemented the structure editor in Java, which provides us with an implementation that is independent of the other components of the ASF+SDF Meta-Environment, and sufficiently fast to cater for interactive use.

The text editor we use is `jedit`, a public domain text editor written in Tcl/Tk. It is easily extendible which allows us to experiment with different lay-outs and user interfaces.

The parsing strategies mentioned in the previous section are implemented as part of the TOOLBUS-script. In the script we decide whether a focus should be parsed, or whether it should first be expanded, and then parsed.

5.4 Future Work

In Section 5.2 we mentioned the need for experimenting with different parsing strategies. By moving the implementation of these strategies out of the structure editor and into the TOOLBUS-script, we hope to create a system that provides us with the flexibility to try out different strategies.

The structure editing specification so far has only been tested on small examples. We need to see how the specification holds up when used with larger, more complex editing sessions.

The text editor that is used currently needs to be replaced with a well known editor. We will try to incorporate emacs into the structure editing system.

Finally, we need to realize a link between the structure editing system and the tree-repository (Section 3). In GSE, when expanding a meta-variable, the editor lists all productions with the same sort as the meta-variable. As the new editing system is completely language independent, it needs to get the list of productions from the tree-repository explicitly.

6 Interpreter

The interpreter or evaluator takes care of rewriting terms given a set of equations. The interpreter rewrites terms in AsFix format using equations in AsFix format. The interpreter was first specified in ASF+SDF, and later this specification was used to make a C implementation.

We will first discuss how the interpreter is activated and which steps are performed before a term is actually rewritten. Then we will discuss the implementation of the interpreter in more detail. Finally, we will discuss some related aspects, such as performance, improvements, etc.

6.1 Activating the Interpreter

The interpreter is activated in the same way as in the old ASF+SDF Meta-Environment, each term editor is extended with a `Reduce`-button. When pushing this button it is first checked whether the interpreter has the appropriate set of rewrite rules available. If not, the equations are retrieved from the tree repository. This is done by retrieving the transitive closure of the import graph from the import repository and sending this set of modules to the tree repository which then returns the set of equations. Finally, the AsFix representation of the term is sent to the interpreter. This is all coded in the TOOLBUS-script.

When the interpreter receives a set of equations it performs some simple transformations on it, for instance, layout is removed and lexicals are transformed into lists of characters. The transformation is performed in order to use the standard list matching mechanism to deal with lexicals. The interpreter stores the equations in a hash table to have fast access to them during rewriting.

The term to be rewritten is also slightly modified, layout is removed and the lexicals are transformed to lists of characters. After rewriting, the result term is again modified: layout is inserted and the lists of characters are translated back into lexicals. The inserted layout is rather arbitrary, to get a better layout of the reduced term it is necessary to adapt this standard unparsing mechanism, see [10] for more details. This term is sent to the TOOLBUS to be displayed in an editor.

The interpreter does not throw away the equations after rewriting. The equations are only thrown away when one of the modules in the specification is modified.

6.2 Implementation

Before discussing the implementation of the interpreter we recall some of the characteristics of the ASF+SDF-formalism, and more specific of ASF itself. The ASF-formalism has the following characteristics:

- The functions are many-sorted.
- The equations may be non left-linear.
- It is possible to use list matching.
- The conditions in the equations may be both positive and negative.
- It is possible to use default equations.
- The evaluation strategy of the equations is based on innermost rewriting.

The main functionality of the interpreter consists of a rewriting machine and a local repository to store equations. This repository is organised as a table, the keys in this table are module names and the values are sets of equations corresponding to the transitive closure of the import graph of the corresponding module. The C implementation of the shared term library takes care of unnecessary duplication of the rewrite rules. In the C implementation of the interpreter the set of equations is stored in a hash-table and the hash key is calculated using the outermost function symbol of the left hand side of an equation and the outermost function symbol (if any) of the first argument of the left hand side of the equation. This improves the efficiency of the rewriting machine enormously, but it influences the semantics as well. In fact this implements a form of syntactic specificity, because when the interpreter is looking for an equation that matches with a term, it first looks for an equation that has the same outermost function symbol (OFS), *and* has the same OFS at the first argument position. If no match can be found, the search is continued for an equation that has the same OFS as the term, but with a variable at the first argument position. This strategy means that equations with a variable at the first argument position are only applied when no other equation is applicable. Finally, when this search also fails the interpreter looks for a default equation that matches with the term.

The rewrite machine itself consists of a collection of recursive functions which have as arguments a set of equations, the term to be rewritten, and an environment in which the instantiated variables are stored.

Recursion is used to implement the backtracking behaviour of list matching in ASF. The instantiation of list variables is done by assigning an “arbitrary” sublist to a list variable. If this does not lead to a successful matching of all variables in the equation or one of the conditions can not be satisfied, another sublist is tried. This process is repeated until either a successful match is found, or all sublists are tried.

ASF+SDF allows the use of conditional equations. The conditions may be positive as well as negative. The current prototype does not allow the introduction of new variables in a negative condition. Furthermore, it is not allowed to introduce new variables on both sides of a positive condition. If new variables are introduced on one side of a positive condition only the other side is rewritten which is then matched against the “variable introducing side” of the condition, leading to new variable bindings.

6.3 Discussion

There are a few open issues with respect to the interpreter. First, the performance of the current version is reasonable. If there is no list matching involved the interpreter performs 6000 rewrite steps per second, the interpreter in the old ASF+SDF Meta-Environment performs 20000 rewrite steps per second for the same specification and term. So, there is at least a factor 3 to be gained.

Second, in the current version no or almost no preprocessing of the specification is performed. A number of preprocessing steps could improve the performance considerably. One obvious preprocessing step is the calculation of which side of a positive condition introduces new variables. Another very effective preprocessing step is the transformation of some forms of list matching into non list matching, e.g., obtaining the head and tail of a list, etc. This can be done because we know the internal data structure for lists in the interpreter. The list transformations are also important when compiling ASF+SDF specifications.

7 Conclusions

In this paper the first prototype of the new ASF+SDF Meta-Environment is discussed. This version of the prototype should be considered as a test case to see whether for instance the TOOLBUS is suited as backbone for the new ASF+SDF Meta-Environment. One of the lessons we learned from the implementation of the old ASF+SDF Meta-Environment is that it is essential to have a flexible and extendible implementation. The ASF+SDF Meta-Environment is first of all a research tool, which means that it should facilitate the testing all kind of new ideas.

Although all sections conclude with a short discussion in which some the future work is described. We left out some general remarks about future work. The ASF+SDF Meta-Environment should provide facilities to compile specifications in order to generate stand-alone environments. Finally, the new ASF+SDF Meta-Environment should provide sophisticated help facilities and demonstration modes with WWW support.

References

- [1] D. Austry. The VTP project: modular abstract syntax specification. Rapports de Recherche 1219, INRIA, Sophia-Antipolis, 1990.
- [2] J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.
- [3] J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press/Addison-Wesley, 1989.
- [4] J. A. Bergstra and P. Klint. The Discrete Time ToolBus. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of *LNCS*, pages 286–305. Springer-Verlag, 1996. Full version: Technical Report P9502, Programming Research Group, University of Amsterdam.
- [5] J. A. Bergstra and P. Klint. The ToolBus coordination architecture. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models (COORDINATION'96)*, volume 1061 of *Lecture Notes in Computer Science*, pages 75–88. Springer-Verlag, 1996. Full version: Technical Report P9408, Programming Research Group, University of Amsterdam.
- [6] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: The System. *SIGPLAN Notices*, 24(2):14–24, 1989. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. The Centaur Tutorial is available at <http://www.inria.fr/croap/centaur/tutorial/main/main.html>.
- [7] M. G. J. van den Brand, J. Heering, and P. Klint. Renovation of the ASF+SDF Meta-Environment—current state of affairs. Technical report, University of Amsterdam, Programming Research Group, 1997. This proceedings.
- [8] M. G. J. van den Brand, P. Klint, P. Olivier, and E. Visser. ATerms: representing structured data for exchange between heterogeneous tools. Technical report, University of Amsterdam, Programming Research Group, 1997. In preparation.
- [9] M. G. J. van den Brand, P. Klint, and C. Verhoef. Core technologies for system renovation. In K.G. Jeffery, J. Král, and M. Bartošek, editors, *SOFSEM '96: Theory and Practice of Informatics*, volume 1175 of *LNCS*, pages 235–255. Springer-Verlag, 1996.
- [10] M. G. J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5:1–41, 1996.
- [11] A. van Deursen. *Executable Language Definitions: Case Studies and Origin Tracking Techniques*. PhD thesis, University of Amsterdam, 1994.
- [12] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific Publishing Co., 1996.

- [13] J. Ellson, E. Koutsofios, and S. C. North. *graphviz* package, release 1.2, 1996. Software package available from <http://www.research.att.com/sw/tools/graphviz/>.
- [14] J. Ellson and S. C. North. TclDG - a Tcl extension for dynamic graphs. In *Proceedings of the fourth annual Tcl/Tk workshop*. USENIX Association, 1996.
- [15] E. R. Gansner, E. Koutsofios, S. C. North, and K. P. Vo. A technique for drawing directed graphs. *IEEE Trans. Softw. Eng.*, 19:214–230, 1993.
- [16] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF — Reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989. Most recent version available at <http://www.cwi.nl/~gipe/>.
- [17] J. Heering and P. Klint. The prehistory of ASF+SDF (1980–1984). In M. G. J. van den Brand, A. van Deursen, T. B. Dinesh, J. F. Th. Kamperman, and E. Visser, editors, *ASF+SDF'95: A Workshop on Generating Tools from Algebraic Specifications*, Technical Report P9504, pages 1–4. Programming Research Group, University of Amsterdam, 1995.
- [18] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *IEEE Transactions on Software Engineering*, 16(12):1344–1351, 1990. Also in: *SIGPLAN Notices*, 24(7):179–191, 1989.
- [19] J. Heering, P. Klint, and J. Rekers. Incremental generation of lexical scanners. *ACM Transactions on Programming Languages and Systems*, 14(4):490–520, 1992.
- [20] INRIA, Rocquencourt. *LeLisp, Version 15.23, reference manual*, 1990.
- [21] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.
- [22] J. W. C. Koorn. *Generating Uniform User-Interfaces for Interactive Programming Environments*. PhD thesis, University of Amsterdam, 1994.
- [23] J. W. C. Koorn and H. C. N. Bakker. Building an editor from existing components: an exercise in software re-use. Report P9312, Programming Research Group, University of Amsterdam, 1993. Available as <ftp://ftp.cwi.nl/pub/gipe/reports/KB93.ps.Z>.
- [24] E. Koutsofios and S. C. North. Drawing graphs with *dot*. Technical report, AT&T Bell Laboratories, Murray Hill, NJ., 1993.
- [25] T. Kuipers. Language independent structure editing using the ToolBus. Master's thesis, Programming Research Group, University of Amsterdam, 1996. Available from <ftp://ftp.wins.uva.nl/pub/programming-research/MasterTheses/Kuipers.ps.gz>.
- [26] E. A. van der Meulen. *Incremental Rewriting*. PhD thesis, University of Amsterdam, 1994.
- [27] S. C. North. Incremental layout in dynaDAG. In *Proc. Graph Drawing '95*, volume 1027 of *Lecture Notes in Computer Science*, pages 409–418. Springer-Verlag, 1996.
- [28] S. C. North and E. Koutsofios. Applications of graph visualization. *Graphics Interface '94*, pages 235–245, 1994.
- [29] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.
- [30] F. Tip. The equation debugger. Master's thesis, University of Amsterdam, Programming Research Group, 1991.
- [31] E. Visser. *ASF+SDF to LaTeX: The User Manual*. Programming Research Group, University of Amsterdam, 1995. <http://adam.fwi.uva.nl/~visser/tolatex/>.
- [32] E. Visser. Executable specification of programming languages. Technical report, Programming Research Group, University of Amsterdam, 1997. In preparation.

- [33] E. Visser. Scannerless generalized-LR parsing. Technical report, Programming Research Group, University of Amsterdam, 1997. (in preparation).