A Framework for Debugging
Olivier, P.A.

Citation for published version (APA):

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

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
The old ASF+SDF Meta-Environment has become a legacy system over the last few years. Most of the work in this thesis has been done in the context of developing a new implementation of the ASF+SDF Meta-Environment, which is based on the latest techniques concerning the coupling of software components, construction of user interfaces and programming languages. The second part of this thesis is centered around two of the key techniques integrated in the new ASF+SDF Meta-Environment: the compilation and debugging of ASF+SDF specifications. We start this second part with a description of this new ASF+SDF Meta-Environment.

The general architecture of the implementation of the new ASF+SDF Meta-Environment is discussed as well as the components which are currently implemented and operational in that environment. Each component is independent of the other components and communicates using the TOOLBUS.

5.1 Introduction

In the beginning of the eighties the design and implementation of the current version of the ASF+SDF Meta-Environment [Kli93] was started. On top of CENTAUR [BCD+89] a programming environment (generator) for writing language definitions in ASF+SDF [HHKR92, BHK89, DDK96] was developed. An overview of these activities can be found in [HK95].

The implementation could be considered a test case for all kinds of ideas concerning the lazy and incremental generation of scanners [HKR92], parsers [HKR90], and term rewriting machines. The development of advanced hybrid editing techniques [Koo94], origin tracking techniques [Deu94], incremental rewriting [Meu94], automatic generation of unparsers [BV96], debugging facilities of term rewriting [Tip91], and the generation of \LaTeX\ code [Vis95] 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 below:

- The user interface is outdated and lacks in user-friendliness. Many operations like
editing or deleting a module require the user to select a specific module. To do this, the user interface presents all modules in a single list. This can be very cumbersome when working on a large specification (more than 100 modules).

- An often heard complaint is: "The editor is too restricted, why is it not emacs- or vi-like?"

- It is not possible to deploy generated tools independent of the ASF+SDF Meta-Environment.

- It is impossible to port the ASF+SDF Meta-Environment to different architectures. The implementation language (LELISP [LeL90]) is essentially obsolete, and only available on a limited number of platforms.

- Most of the implementation is centered around the tree formalism VTP [Aus90]. This formalism is difficult to use, and the connection between LELISP and VTP is cumbersome.

- 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 a new ASF+SDF Meta-Environment can be found in [BHK97].

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 [Koo94] by Emacs and Epoch in 1992 [KB93]. However, it proved that it was impossible to manage the interaction between the different tools. This initiated the development of the TOOLBUS software interconnection architecture introduced in Chapter 2. The 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 was based was becoming obsolete made a "from scratch" approach even more urgent. In this chapter we discuss a first prototype of the new ASF+SDF Meta-Environment based on the TOOLBUS. This prototype offers an extendible infrastructure to experiment with various designs.
In the rest of this chapter the most important components of the new ASF+SEF Meta-Environment are presented. In Section 5.3 the architecture of the new ASF+SEF Meta-Environment is discussed. Section 5.4 describes the tree-repository to store ASF+SEF modules and terms, furthermore the tree representation format is briefly discussed. The user interface is discussed in Section 5.5, the structure editor in Section 5.6, the parser and parser generator in Section 5.7, the compiler is introduced in Section 5.8 and the interpreter is discussed in Section 5.9.

5.2 Examples of ASF+SEF Specifications

We don't give examples of ASF+SEF specifications here, but refer the interested reader to Figures 7.1, 7.2, 8.1, 8.2, 8.5, and 8.9 in later chapters.

5.3 General Architecture

The architecture of the new ASF+SEF Meta-Environment is depicted in Figure 5.1. This figure is a snapshot of the current state of the prototype, see Table 5.1 for a more detailed list of components and their implementation languages. The unparsen generator is currently only available as a stand-alone tool, but will be integrated in the ASF+SEF Meta-Environment in the future.

Table 5.1 gives an overview of all currently available components in the prototype. For
5.1: Components of the new ASF+SDF Meta-Environment

<table>
<thead>
<tr>
<th>Component</th>
<th>Specification Language</th>
<th>Implementation Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATerms</td>
<td>ASF+SDF</td>
<td>C, Java, Tcl, Emacs lisp</td>
</tr>
<tr>
<td>AsFix</td>
<td>ASF+SDF</td>
<td>C</td>
</tr>
<tr>
<td>TOOLBUS</td>
<td>ASF+SDF</td>
<td>Tcl/Tk, TclDot</td>
</tr>
<tr>
<td>User Interface</td>
<td></td>
<td>Emacs lisp</td>
</tr>
<tr>
<td>Text editor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Structure editor</td>
<td>ASF+SDF</td>
<td>Java</td>
</tr>
<tr>
<td>Parse table interp.</td>
<td>ASF+SDF</td>
<td>C</td>
</tr>
<tr>
<td>Parse table gener.</td>
<td>ASF+SDF</td>
<td>C</td>
</tr>
<tr>
<td>Tree repository</td>
<td>ASF+SDF</td>
<td>C</td>
</tr>
<tr>
<td>ASF+SDF Compiler</td>
<td>ASF+SDF</td>
<td>ASF+SDF</td>
</tr>
<tr>
<td>Interpreter</td>
<td>ASF+SDF</td>
<td>C</td>
</tr>
<tr>
<td>Unparser Generator</td>
<td>ASF+SDF</td>
<td>ASF+SDF</td>
</tr>
</tbody>
</table>

Table 5.1: Components of the new ASF+SDF Meta-Environment

each component it is listed whether this component is specified and in which language it is implemented. The first two components in the table are in fact datatypes, which are used by all other components.

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 cannot 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 5.4.1 and Chapter 6 for more details).

5.4 The tree-repository

All components except the debugger manipulate some form of (abstract) syntax trees. It is the responsibility of the tree-repository to store these trees. Before discussing the implementation details of the tree-repository, we discuss the general format of all stored and exchanged information.

5.4.1 Tree Representation

In the old ASF+SDF Meta-Environment the abstract syntax trees are represented by means of VTP (Virtual Tree Processor) [Aus90] offered by CENTAUR [BCD"89]. There are two
problems connected to VTP: it is hard to learn programming in VTP, and VTP does not offer enough facilities to prevent illegal access to constructed trees. The latter drawback caused 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 (see Chapter 6).

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

\[
\begin{align*}
\text{constant} & \quad abc \\
\text{numeral} & \quad 123 \\
\text{literal} & \quad "\text{abc}" \text{ and } "123" \\
\text{list} & \quad [1, \text{ "abc"}, 3 \text{ and [1, 2, [3,4], 5]} \\
\text{function} & \quad f("\text{abc}") \\
\text{annotation} & \quad f(123)\{\text{color("red"), path([0,2,1])}\}
\end{align*}
\]

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 have been formally specified in ASF+SDF and we have build library implementations in C and Java that are all based on this formal specification. These libraries are used in all components to manipulate terms.

### 5.4.2 Representing Syntax Trees: AsFix

The ATerm data type proves to be a powerful and flexible mechanism to represent syntax trees. By defining an appropriate set of function symbols parse trees and abstract syntax trees can be represented for any language or formalism. As an example, we present AsFix, a parse tree format for ASF+SDF.

**AsFix** The AsFix format (ASF+SDF Fixed format) is an incarnation of ATerms for representing ASF+SDF.

Using AsFix, each module or term is represented by its parse tree which contains both the syntax rules used and all original layout and comments. In this way, the original source text can be reconstructed from the AsFix representation, thus enabling transformation tools to access and transform comments in the source text. Since the AsFix representation is self-contained (all grammar information needed to interpret the term is also included), one can easily develop tools for processing AsFix terms which do not have to consult a common database with grammar information. Examples of such tools are a (structure) editor or a rewrite engine.

AsFix is defined by an appropriate set of function symbols for representing common constructs in a parse tree. These function symbols include the following:
• \( \text{prod}(T) \) represents production rule \( T \).
• \( \text{appl}(T_1, T_2) \) represents applying production rule \( T_1 \) to the arguments \( T_2 \).
• \( 1(T) \) represents literal \( T \).
• \( \text{sort}(T) \) represents sort \( T \).
• \( \text{lex}(T_1, T_2) \) represents (lexical) token \( T_1 \) of sort \( T_2 \).
• \( w(T) \) represents white space \( T \).
• \( \text{attr}(T) \) represents a single attribute.
• \( \text{attrs}(T) \) represents a list of attributes.
• \( \text{no-attrs} \) represents an empty list of attributes.

The following context-free syntax rules (in SDF [HHKR92]) are necessary to parse the input sentence true or false.

\[
\begin{align*}
\text{sort} & \quad \text{Bool} \\
\text{context-free syntax} & \\
\text{true} & \rightarrow \text{Bool} \\
\text{false} & \rightarrow \text{Bool} \\
\text{Bool or Bool} & \rightarrow \text{Bool} \text{ (left)}
\end{align*}
\]

The parse tree below gives the AsFix representation for the input sentence true or false.

\[
\text{appl}(\text{prod}([\text{sort}(\text{"Bool"}), \text{l("or")}, \text{sort}(\text{"Bool"})], \text{sort}(\text{"Bool"}), \\
\text{attrs}([\text{attr(\"left")}]))) \\
\quad \text{[appl(\text{prod}([\text{l("true")}], \text{sort}(\text{"Bool"}), \text{no-attrs}), [\text{l("true")}])),} \\
\quad \text{w(\" "), l("or"), w(\" "),} \\
\quad \text{appl(\text{prod}([\text{l("false")}], \text{sort}(\text{"Bool"}), \text{no-attrs}), [\text{l("false")}]])}
\]

Two observations can be made about this parse tree. First, this parse tree is an ordinary ATerm, and can be manipulated by all ATerm utilities in a completely generic way.

Second, this parse tree is completely self-contained and does not depend on a separate grammar definition. It is clear that this way of representing parse trees contains much redundant information. Therefore, both maximal sharing and BAF (as presented in Chapter 6) are essential to reduce their size.

The annotations provided by the ATerm data type can be used to store auxiliary information like position information derived by the parser or font and/or color information needed by a (structure) editor. This information is globally available but can be ignored by tools that are not interested in it.
5.4.3 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, such as its import section or its equations. It is also possible to compute the transitive closure of import relations of a module. The tree-repository is implemented as a table with the module name and term name as key and the AsFix representation as value.

5.4.4 Discussion

The information stored in the tree-repository could be extended with all kinds of extra information such as the size of the file used for the persistent storage of each term, its creation date, etc. Furthermore, the tree-repository should provide a sophisticated querying mechanism as described in [BKV96]. Such a mechanism can be used to locate specific definitions of sorts, lexical and context-free syntax rules, and the like.

5.5 User Interface

Figure 5.2 contains a screeendump 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.

The user interface of the new ASF+SDF Meta-Environment is built around a visual representation of the import graph of a given ASF+SDF specification. 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 visualization 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.
- Delete the module from the repository.
- Request additional information about the module.
- 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-hand-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.
- Save the import graph in various graphics formats.

5.5.1 Implementation

The user interface is implemented in Tcl/Tk [Ous94] and TclDot: an extension for Tcl that incorporates the directed graph facilities of dot [KN93] 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 [GKNV93].

TclDot is part of the graphviz package [NK94, EKN96]: a set of graph drawing tools for
Unix or MS-Windows from AT&T/Lucent Technologies.

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.

The user interface is notified by the TOOLBUS of state changes in the list of modules, or
in the import relations of a module so it can updates its visual representation of the import
graph.

5.5.2 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 TclDot 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 undesirable for a user interface since it can be confusing
and the user needs to familiarize himself with a new structure. A new version of TclDot,
called TclIDG, supports so called dynamic graphs [Nor96, EN96]. 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 TclIDG 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 (e.g., 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.

5.6 Editors

The structure editing system in the new implementation provides roughly the same function-
ality as the Generic Structure Editor (GSE [Koo94]). There are, however a number of
differences, both in the external and the internal behavior.
5.6.1 Internal Behavior

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 behavior (a Tool-Bus interface) [Kui96]. 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 $\alpha$ is a syntactically correct string in the language $L$, and we have a parser $\Pi_L$ 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 $\Pi_L(pp(t)) = t$.

5.6.2 External Behavior

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:

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

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

```plaintext
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.

However, 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.

5.6.3 Implementation

As stated above, the editing system has been implemented as two separate tools. The structure editor was first specified in ASF+SDF, and after prototyping it in Java it was finally implemented in C for performance reasons. The text editor is currently based on emacs, and all ASF+SDF Meta-Environment specific functionality is coded in emacs lisp. The interface of the text editor is flexible enough to add support for other editors in the future.

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.

5.6.4 Future Work

In Section 5.6.2 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.
Last but not least, we need to realize a link between the structure editing system and the tree-repository (Section 5.4). 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.

5.7 Parsing

The user definable syntax of ASF+SDF makes the parsing technology in the new ASF+SDF Meta-Environment very important. We need a parser that can be instantiated with each language over which we want to parse terms. To do this, we have a parser generator that generates a parse table given the syntax of a language. These parse tables are used to instantiate the parser (SGLR) to parse terms over the syntax.

5.7.1 Parser Generator

From an SDF syntax definition a parse table is generated for later use by SGLR (see Section 5.7.2). Note that the new parser generator is no longer based on incremental and lazy generation techniques [HKR90]. The generation process consists of two distinct phases. In the first phase, the SDF definition is normalized to an intermediate, rudimentary, formalism: Kernel-SDF. In the second phase this Kernel-SDF is transformed to a parse table.

*Grammar Normalization* The grammar normalization phase consists of several steps and concludes by producing a Kernel-SDF definition. The most important steps are the following:

- A modular SDF specification is transformed into a flat specification.
- Lexical grammar rules are transformed to context-free grammar rules.
- Priority and associativity definitions are transformed to a list of pairs, where each pair consists of two production rules for which a priority or associativity relation holds. The transitive closure of the priority relations between grammar rules is made explicit in these pairs.

*Parse Table Generation* The actual parse table is derived from the Kernel-SDF definition. To do so, a rather straightforward SLR(1) approach is taken. However, possible shift/reduce or reduce/reduce conflicts are unproblematic, and can hence simply be stored in the table. Some extra calculations are then performed in order to reduce the number of conflicts in the parse table. Based on the list of priority relation pairs the table is filtered; see [KV94] for more details. The resulting table contains a list of all Kernel-SDF production rules, a list of states with the actions and gotos, and a list of all priority relation pairs. The parse table is represented as an ordinary ATerm, see Chapter 6.
5.7.2 Scannerless Generalized LR Parsing (SGLR)

Even though parsing is often considered a solved problem in computer science, every now and then new ideas and combinations of existing techniques pop up. SGLR (Scannerless Generalized LR) parsing is a striking example of a combination of existing techniques that results in a remarkably powerful parser.

*Generalized LR Parsing for Context-Free Grammars*  LR parsing [ASU86] is a well-known parsing technique used in many well-known implementations, e.g. LEX/YACC [LS86, Joh86] and FLEX/BISON (the GNU counterpart of LEX/YACC). LR parsers are based on the shift/reduce principle; a (conflict-free) LR(k) (k \( \geq 0 \)) parse table, containing actions and gotos, is used. A conventional LR parser consists of a scanner, that splits the input stream into tokens, and a parser that processes the tokens and either generates error messages or builds a parse tree.

LR parsing restricts the class of languages that can be recognized to LR(k) (k \( \geq 0 \)). Although all kinds of subtleties exist (such as LALR(k)), these are of no relevance to this discussion.

The ability to cope with arbitrary context-free grammars is of major importance if one wishes to allow a modular syntax definition formalism. Due to the fact that LR(k)-grammars are not closed under union, a more powerful parsing technique is required. Generalized LR-parsing [Tom85, Rek92] (GLR-parsing) is a natural extension to LR-parsing, from this perspective. The saving grace of GLR-parsing lies in the fact that it does not require the parse table to be conflict-free. Allowing conflicts to occur in parse tables, GLR is equipped to deal with arbitrary context-free grammars.

If, while parsing, a conflict in the parse table occurs, the parse stack is split; every possible continuation gets its own parse stack. These stacks are processed in parallel, but the stacks are synchronized when shifting input tokens. Stacks sharing the same state are merged at that point.

The parse result, then, might not consist of a single parse tree; generally, a forest consisting of an arbitrary number of parse trees is yielded. Ambiguity produces multiple parse trees, each of which embodies a parse alternative. In case of an LR(1) grammar, the GLR algorithm collapses into LR(1), and exhibits similar performance characteristics. As a rule of thumb, the simpler the grammar, the closer GLR performance will be to LR(1) performance.

*Eliminating the Scanner*  The GLR parser as it is found in the old ASF+SDF Meta-Environment uses a scanner, just like any conventional LR(k) parser. The use of a scanner in combination with GLR parsing leads to a certain tension between scanning and parsing. In a number of situations the scanner has several ways of recognizing a string of characters: there is a so-called lexical ambiguity\(^1\). At that point, the scanner has to come to some decision; later, when parsing the tokens as output by the scanner, the selected tokenization might turn out not to be what the parser expected, causing the parse to fail.

\(^1\)Consider the well-known examples of the range operator vs. the real numbers in Pascal.
Scannerless GLR parsing [Vis97] solves this problem by unifying scanner and parser. In other words: the scanner is eliminated by simply considering all elementary input symbols to be input tokens for the parser. Each character becomes a separate token, and ambiguities on the lexical level are dealt with by the GLR algorithm. To accomplish this level of integration, the lexical syntax rules are transformed into context-free syntax rules.

5.7.3 Implementation

The parse table generator  The parser generator consists of two phases as described in Section 5.7.1. Both phases where initially specified in ASF+SDF, and compiled to C. This method of implementation proved too inefficient for the second phase, the generation of the actual parse table. Consequently, we have reimplemented this phase in C. A good illustration of the performance gained this way is the time it now takes to generate the parse table for COBOL. Before the reimplemention this generation would take about 78 minutes. Using the C implementation this time was reduced to about 80 seconds. Only about 10 of these 80 seconds are used by the second phase. It is clear that any new gains must come from the first phase, for instance by also reimplementing it in C.

The parser  For efficiency reasons, SGLR is implemented in C. Via the TOOLBUS, SGLR can be instantiated with parse tables. After this instantiation, strings can be parsed using one of these parse tables. The result can be one of three things:

- A parse tree after a successful parse without ambiguities.
- A parse forest after a successful parse where ambiguities were encountered.
- A parse error when the input string did not form a valid sentence over the input syntax.
- A cycle detection error when the input syntax contained cycles. Not all of these cycles can be detected statically by looking at the input syntax, so cycle detection errors can be encountered when parsing.

5.7.4 Discussion

The current parser is not very good in error diagnosis, and error recovery is lacking completely. An elegant solution for handling ambiguities is also needed.

A second problem with the current implementation is related to the modular nature of ASF+SDF specifications. Because the syntax definitions are also modular, a specification actually consists of a hierarchical structure of language specifications. In the current situation, for each module a different parse table is needed that represents the syntax of that module and all of its imported modules.

Clearly, each of these parse tables has a lot in common with the parse tables of its imported modules. More research is needed to add some kind of modularization support to both the parse table generator and the parser, in order to reduce this redundancy.
5.8 Compiler

The new ASF+SDF Meta-Environment provides two mechanisms for executing specifications. The interpreter is used for incremental development of specifications, and is discussed in the next section. The performance of the interpreter is not good enough for industrial strength applications, but the turnaround time for changes in specifications is excellent. When a specification is finished, the ASF+SDF compiler can be used to generate C code that efficiently rewrites terms according to the specification.

The ASF+SDF compiler will be discussed in full detail in Chapter 7.

5.9 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. A first version of the interpreter was specified in ASF+SDF. Based on this specification an efficient version was handcrafted in C.

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.

5.9.1 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. After pushing this button a check is performed whether the interpreter has the appropriate set of rewrite rules available. If not, all equations of the modules in the transitive closure of the import graph are retrieved from the module repository.

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 into 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 [BV96] 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.
5.9.2 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 ATerm 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 behavior 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.

5.9.3 Discussion

There are a few open issues with respect to the interpreter. First, although the performance of the current version is reasonable, we feel it can still be improved. When no list matching is involved, the current version is about twice as fast as the old ASF+SDF Meta-Environment. However, performance decreases drastically when matching lists.

When looking for ways to improve the performance of the interpreter, it is important to keep in mind that it will also be used as a tool to experiment with aspects of the execution of ASF+SDF specifications. This means that the implementation must be kept open and easy to comprehend.

One of the areas in which we are currently experimenting is debugging of ASF+SDF specifications. In Chapter 8 we show how the interpreter can be connected to the TIDE system described in Chapter 4.

It will indeed prove challenging to keep improving the performance of the interpreter without sacrificing extensibility.

One technique that will allow us to significantly gain performance, is the use of preprocessing of specifications. 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. These kinds of list transformations are also important when compiling ASF+SDF specifications (see Chapter 7).

5.10 Debugging issues

Debugging in the context of the new ASF+SDF Meta-Environment has three dimensions:

1. Debugging of the new ASF+SDF Meta-Environment itself
2. Debugging of ASF+SDF specifications
3. Debugging of programs written in languages specified in ASF+SDF

Our work presented in the first part of this thesis can be used to cover all three dimensions.
The ASF+SDF Meta-Environment is a sophisticated distributed system that is based on the TOOLBUS. The TIDE system discussed in Chapter 4 can therefore be used directly to debug the ASF+SDF Meta-Environment. The individual components can be debugged as long as they are executed using a language implementation that is supported by TIDE. Luckily, this is the case for almost all components in the ASF+SDF Meta-Environment, as the implementation languages are C, Java, Tcl/Tk, and ASF+SDF. Only the text editor, which is written in emacs lisp, cannot be debugged using TIDE at this time.

The latter two dimensions, debugging of ASF+SDF specifications and debugging of programs written in languages that are specified in ASF+SDF, are discussed in Chapter 8.

5.11 Conclusions

In this chapter 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 of all kinds of new ideas.