Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

Development of Parsing Tools for CASL using Generic Language Technology

M.G.J. van den Brand, J, Scheerder

Software Engineering (SEN)

SEN-R0011 May 31, 2000

Report SEN-R0011
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Development of Parsing Tools for CASL using Generic Language Technology

Mark van den Brand
Mark.van.den.Brand@cwi.nl

Jeroen Scheerder
Jeroen.Scheerder@cwi.nl

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT

An environment for the Common Algebraic Specification Language CASL consists of several independent tools. A number of CASL tools have been built using the algebraic specification formalism ASF+SDF and the ASF+SDF Meta-Environment. CASL supports user-defined syntax which is non-trivial to process: ASF+SDF offers a powerful parsing technology (Generalized LR). Its interactive development environment facilitates rapid prototyping complemented by early detection and correction of errors. A number of core technologies developed for the ASF+SDF Meta-Environment can be reused in the context of CASL. Furthermore, an instantiation of a generic format developed for the representation of ASF+SDF specifications and terms provides a CASL-specific exchange format.

1999 ACM Computing Classification System: D.2.6, D.2.m, D.3.1, D.3.4, E.1
Keywords and Phrases: Parsing; User-defined syntax; CASL
Note: To appear in *Proceedings of Workshop on Algebraic Development Techniques (WADT'99)*, LNCS 1827, 2000.
Note: Work carried out under project SEN-1.4, ASF+SDF.

1. INTRODUCTION

CASL (Common Algebraic Specification Language) [11] is a new algebraic specification formalism developed as part of the Common Framework Initiative (COFI). It is a generic algebraic specification formalism incorporating features of most existing algebraic specification languages. To complement the CASL formalism itself, a set of tools to support development of CASL specifications is planned. For this purpose, existing tools and technologies will be reused wherever possible.

The algebraic specification formalism ASF+SDF [13] and the ASF+SDF Meta-Environment [24] have been deployed to prototype CASL's concrete syntax. The user-defined (also known as mixfix) syntax of CASL calls for a two-pass approach. In the first pass, the skeleton of a CASL specification is derived in order to extract user-defined syntax rules. In a second pass these syntax rules are used to parse the expressions using them. ASF+SDF offers the advantages of its underlying powerful parsing technology (Generalized LR parsing) and its interactive development environment that enables early detection of errors, such as lexical and syntactic ambiguities, in the syntax definition of a language. The GLR parsing technology eliminates the need to worry about parse table conflicts, which are especially annoying for languages whose syntax definition is still 'on the move'.

At present, two versions of the ASF+SDF Meta-Environment coexist. One is described in [24]; we refer to this environment as the *traditional* one. The other is a next-generation ASF+SDF Meta-Environment [9] under active development; we refer to this enviroment as *current*. The main difference between traditional and current environments is architectural organization: the former is monolithic, the latter consists of a suite of independent components communicating via a software coordination architecture [4]. The ASF+SDF Meta-Environment uses a common exchange format, ASFIX, to represent parse trees for specifications and terms. It

contains all relevant information, including applied production rules, layout, and comments. ASFIX stands for ASF+SDF fixed format; it is based on ATERMS [8].

Components from the current ASF+SDF Meta-Environment can be (re)used in the context of other systems – such as a CASL environment – in a flexible way.

This paper discusses the following topics:

- Various techniques involved in implementing a CASL parser using ASF+SDF language technology.
- A mapping, CASFIX, from the concrete syntax of CASL to an abstract syntax in ATERMS.
- Treatment of user-defined annotations.

1.1 Overall Architecture of a CASL Parsing Environment

The overall architecture of a full CASL parser, including user-defined syntax, based on ASF+SDF technology is depicted by Figure 1.

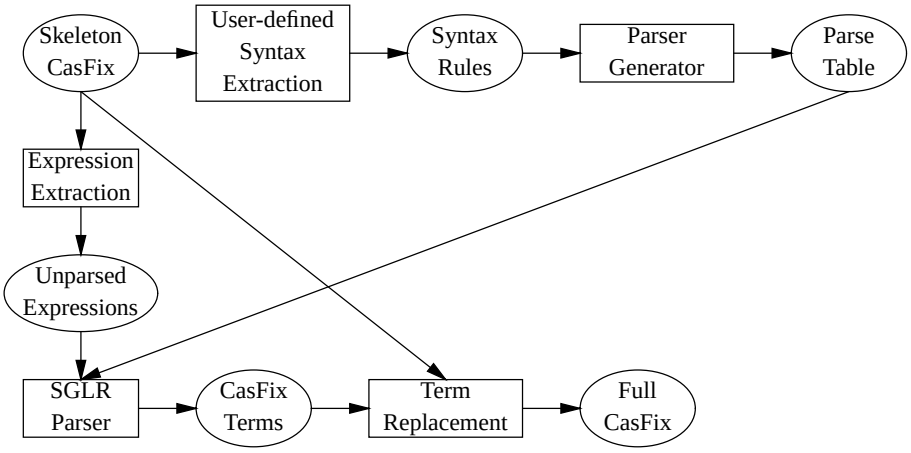

Figure 1: CASL Parser Architecture

'Skeleton CASFIX' is the abstract syntax tree of a CASL specification in which the user-defined syntax expressions are not yet parsed (see Section 2.1). The `SyntaxExtraction` tool extracts definitions of user-defined syntax from this abstract syntax tree, flattens parsed CASL specifications and retrieves all visible sorts, operator definitions, and variables. This tool is based on the ATERM library and implemented in C. The syntax information obtained this way is fed to the `ParserGenerator` tool (details in Section 3.1) to generate a parse table.

The expressions that are yet to be parsed, also present in the abstract syntax tree, are extracted by the `ExpressionExtraction` tool and consecutively parsed using the SGLR parser (discussed in Section 3.2), instantiated with the corresponding parse table that was just generated. In case of a successful parse a syntax tree is constructed in a representation (CASFIX, described in Section 4) tailored specifically to the needs of CASL.

Finally, the `TermReplacement` tool, which is the inverse of `ExpressionExtraction`, substitutes the parsed CASL term for its unparsed counterpart. This tool is also based on the ATERM library and implemented in C.

1.2 Related Work

Related work can be divided into two areas. The first is related to tackling user-defined syntax in general and the second is related to parsers developed for CASL.

Other Approaches to Parsing User-Defined Syntax It is impossible to list all systems that deal with formalisms allowing user-defined syntax exhaustively. We restrict ourselves to those systems that arguably relate to the work on CASL.

The first system we mention is the traditional ASF+SDF Meta-Environment. The algebraic specification formalism ASF+SDF [3, 17, 13] is used, notably for language prototyping. By means of an SDF definition, the lexical and context-free grammar of a language are specified, whereas the ASF definition serves to define the semantics. The SDF part corresponds to signatures in ordinary algebraic specification formalisms. However, syntax is not restricted to plain prefix notation: arbitrary context-free grammars can be defined. The syntax defined in the SDF-part of a module is available for the definition of equations. Therefore, equation syntax is user-defined. In a two-pass approach the SDF definition is parsed, obtaining the definitions necessary for parsing the equations. The parsing technology used to deal with this user-defined syntax is GLR [33, 31, 34].

The second system, ASD [12], is closely related to ASF+SDF. Again a two-pass approach is used to generate parsers for the user-defined syntax occurring in Action Semantics specifications [29]. In the first pass the user-defined syntax definitions are collected. These are then processed by an ASF+SDF specification and transformed into another SDF definition. The resulting definition is then used to parse the expressions that may contain user-defined syntax.

The third system is CIGALE [35], a parser for ASSPEGIQUE [5]. CIGALE supports incremental grammar definition, and offers a great flexibility in syntax. It is not derived from standard techniques like Earley [15] or LR [1]. At its core lies a restrictive backtracking algorithm.

Finally, we mention the parser used in the ELAN system [6]. ELAN is a specification language based on rewriting logic with additional features, notably strategies. The parser used in the ELAN system is based on the Earley parsing technology. One drawback of ELAN is the way constraints imposed by the Earley parser are visible in the ELAN specification; for example, rewrite rules are grouped according to the 'sort' of the left- and right-hand side in the syntax definition.

Other CASL Parsers Several efforts towards implementing a parser for CASL have been undertaken. The following ones are currently available:

- The parser developed by Frederic Voisin. This parser is based on SYNTAX [7], a LEX/YACC-style parser generator. This CASL parser is comparable to the ASF+SDF-based skeleton parser described in Section 2.1; it also does not parse expressions in user-defined syntax.
- The HOL-CASL parser [28] is a full two-pass CASL parser. It performs skeleton parsing, using a functional implementation of LEX/YACC; user-defined syntax is then parsed using the Isabelle parser, which is based on the Cocke-Younger-Kasami algorithm [19]. The CYK-parser is not very efficient; GLR usually performs better for the most common cases. Except for specifications of very limited size, the difference in performance can on average be expected to be of significant practical relevance.
- The CASL parser by Bastian Kleineidam[1], developed using JAVA[2] and JAVACC[3] at the Max-Planck-Institut für Informatik (Saarbrücken).
- A parser frontend has been developed in order to be able to employ the INKA theorem prover[4] [20] for CASL specifications. This parser is based on LEX/YACC technology. The INKA theorem prover derives proof obligations from CASL specifications, and assists in finding proofs.

2. DEVELOPING A CASL GRAMMAR USING ASF+SDF TECHNOLOGY

Generic language technology such as ASF+SDF can be applied at several stages when developing tools for CASL. The first stage, probably the most obvious one, is set at the level of defining the concrete syntax of CASL.

[1] http://www.mpi-sb.mpg.de/~calvin/
[2] http://www.javasoft.com
[3] http://www.suntest.com/javacc/
[4] http://www.dfki.de/vse/systems/inka/

Using the EBNF definition provided by the CASL language summary [11] an SDF definition[5] has been developed. Figure 2 shows a tiny fragment of this SDF definition. It should be noted that EBNF can be mapped to SDF in a straightforward manner, requiring no complex grammar transformations whatsoever.

```
"{" "}"      -> Basic-Spec
Basic-Item+ -> Basic-Spec

Sig-Items                                          -> Basic-Item
"free" Datatype-S { Datatype-Decl ";"}+ Opt-Semi     -> Basic-Item
"generated" Datatype-S {Datatype-Decl ";"}+ Opt-Semi -> Basic-Item
"generated" "{" Sig-Items+ Opt-Semi "}" Opt-Semi     -> Basic-Item
Var-S {Var-Decl ";"}+ Opt-Semi                       -> Basic-Item
Var-S {Var-Decl ";"}+ "."
  {Labelled-Formula "."}+ Opt-Semi                   -> Basic-Item
Axiom-S {Labelled-Formula ";"}+ Opt-Semi             -> Basic-Item
```

Figure 2: Fragment of the SDF Definition of CASL

This SDF definition does not cope with every aspect, particularly not with user-defined syntax; see Section 3.

2.1 Skeleton Parser

Based on the EBNF definition in [11] the concrete syntax of CASL has been defined. Furthermore, a mapping from concrete syntax to abstract syntax (CASFIX, see Section 4), has been defined in ASF+SDF. Using the parser generator that is part of the ASF+SDF Meta-Environment, a stand-alone parser for CASL has been generated (see Figure 3). At this point it does not yet handle user-defined syntax. This parser, therefore, can be

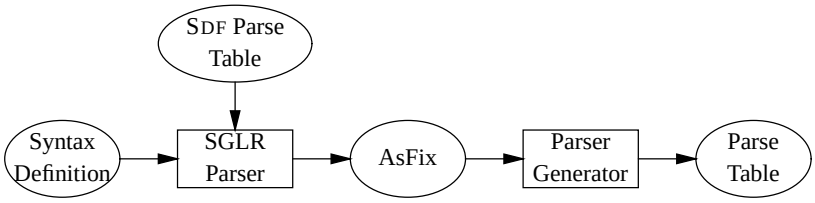

Figure 3: CASL Parse Table Generation

viewed as a skeleton parser that can be invoked as a first-phase parser (shown in Figure 4). By using this parser and applying the CASFIX mapping, the concrete syntax for CASL specifications is obtained. Expressions that

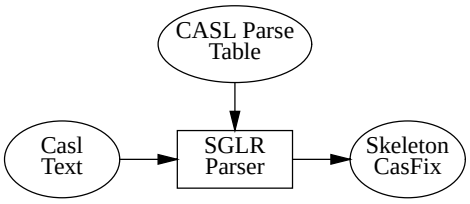

Figure 4: CASL Skeleton Parser

may contain user-defined syntax are not parsed, and are represented as unstructured character sequences, to be analyzed in more detail at some later point, in CASFIX.

Figure 5 provides an exemplary CASL specification; Figure 6 reveals its CASFIX representation. Note the `unparsed-formula`s in Figure 6 which represent the expressions yet to be parsed later on, as discussed in Section 3.

3. PARSING CASL USING ASF+SDF TECHNOLOGY

There are several ways of implementing a CASL parser by means of ASF+SDF technology. One approach is extensively studied and described in [37]. ASF equations are used to parse expressions in user-defined syntax. This approach has proved feasible, but too slow for most practical purposes. Another approach is described in [12] where ASF+SDF is used to build tools for Action Semantics [14]. A parser for Action Semantics [29]

[5] Available at `http://www.cwi.nl/~markvdb/cofi/zcasl-sdf2.html`.

```
spec Monoid =
  sort Elem
  ops n : Elem;
      __*__ : Elem *Elem -> Elem, assoc, unit n
%%  Alternatively, just specify the corresponding axioms:
  vars x,y,z : Elem
  .
    n*x=x
  .
    x*n=x
  .
    (x*y)*z=x*(y*z)
```

Figure 5: Monoid, a small CASL Fragment

exemplifies this. We propose an alternative approach, based on reusing components of the current ASF+SDF Meta-Environment, most notably the parse table generator and the SGLR parser.

The approach in which CASL specifications are parsed using ASF+SDF technology is directly analogous to the approach taken in the ELAN project [6] for parsing user-defined syntax.

3.1 Parser Generator

The parser generator, part of the current ASF+SDF Meta-Environment, is one of the components that can be (re-)used to generate parse tables for user-defined syntax in CASL.

It generates parse tables, suitable for later perusal by the SGLR parse table interpreter (see Section 3.2) from SDF syntax definitions. Unlike previous ASF+SDF parser generation technology, the current implementation does not use incremental and lazy techniques [18].

The process of generating parse tables consists of two distinct phases. In the first one 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, which derives a Kernel-SDF definition, consists of the following steps:

- A modular SDF specification is transformed into a flat specification.
- Lexical grammar rules are transformed to context-free grammar rules.

```
spec-defn(
 SIMPLE-ID(WORDS("Monoid")),
 genericity(params(SPEC*([])),imports(SPEC*([]))),
 BASIC-SPEC(basic-spec(
  BASIC-ITEMS*([
   SIG-ITEMS(sort-items(SORT-ITEM+([
                        sort-decl(SORT+([TOKEN-ID(TOKEN(WORDS("Elem")))]))]))),
   SIG-ITEMS(op-items(OP-ITEM+([
    op-decl(OP-NAME+([ID(TOKEN-ID(TOKEN(WORDS("n"))))]),
            total-op-type(sorts(SORT*([])),
             TOKEN-ID(TOKEN(WORDS("Elem")))),
            OP-ATTR*([])),
    op-decl(OP-NAME+([
             ID(MIXFIX-ID(token-places(TOKEN-OR-PLACE+([
                                        "__",TOKEN(SIGNS(["*"])),"__"])))])),
            total-op-type(sorts(SORT*([TOKEN-ID(TOKEN(WORDS("Elem"))),
                                       TOKEN-ID(TOKEN(WORDS("Elem")))])))
             TOKEN-ID(TOKEN(WORDS("Elem")))),
            OP-ATTR*([associative,unit-op-attr("n")])))),
    local-var-axioms(
     VAR-DECL+([var-decl(VAR+([var(WORDS("x")),var(WORDS("y")),var(WORDS("z"))]),
                         TOKEN-ID(TOKEN(WORDS("Elem"))))]),
     FORMULA+([unparsed-formula(" n * x = x "){label(ID*([])},
               unparsed-formula(" x * n = x "){label(ID*([])},
               unparsed-formula(" ( x * y ) * z = x * ( y * z ) "){label(ID*([])}
              ]))))))
```

Figure 6: Monoid in CASFIX: Skeletal

- Priority and associativity definitions are transformed to lists 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 straightforward SLR(1) approach is taken. However, shift/reduce or reduce/reduce conflicts are not considered problematic, and are simply stored in the table. Some extra calculations are consequently performed to reduce the number of conflicts in the parse table. Based on the list of priority relation pairs the table is filtered; see [25] 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.

3.2 Scannerless Generalized LR Parsing

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 The ability to cope with arbitrary context-free grammars is of pivotal 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 [33, 31] (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. One of the advantages of this approach in the context of CASL is the simple, direct, mapping from the definition of the concrete syntax in EBNF into an SDF definition of same.

The parse result, then, might not be a single parse tree; in principle, 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 in the traditional 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. The scanner may sometimes have several ways of splitting up the input: a so-called lexical ambiguity occurs. In CASL, compound identifiers like '`s[t[u]]`' exhibited this problem (forcing one to modify the specification to something like '`s[t[u]␣]`'). The CASL syntax was adapted to get rid of this unwanted need to modify valid specifications. In case of lexical ambiguities, a scanner must take some decision; at a later point, when parsing the tokens as offered by the scanner, the selected tokenization might turn out to be not quite what the parser expected, causing the parse to fail.

Scannerless GLR parsing [34] solves this problem by unifying scanner and parser. In other words, the scanner is eliminated by simply considering all elementary input symbols (e.g. octets) as input tokens for the parser. Each character becomes a separate token, and ambiguities on the lexical level are dealt with by the GLR algorithm. This way, in a scannerless parser lexical and context-free syntax are integrated into a single grammar, describing the defined language entirely and exhaustively. Neither knowledge of the (usually complex) interface between scanner and parser nor knowledge of operational details of either is required for an understanding of the defined grammar. In the case of CASL, which is a language of which both the lexical as well as the context-free syntax are rather complex, scannerless parsing technology proved beneficial.

3.3 Parsing User-Defined Syntax

The CASFIX representation (see Section 4) of a CASL specification contains all information necessary for further analysis. Operator definitions, defined sorts, variables and the like can be extracted. Using this additional information a new, enriched, parse table that includes the user-defined syntax can then be constructed.

```
sorts Elem
syntax
  <START>[\256]                                        -> <Start>
  <LAYOUT?-CF><Elem-CF><LAYOUT?-CF>                    -> <START>
  "n"                                                  -> <Elem-CF>
  <Elem-CF><LAYOUT?-CF>"*"<LAYOUT?-CF><Elem-CF>        -> <Elem-CF> {assoc}
  "("<LAYOUT?-CF><Elem-CF><LAYOUT?-CF>")"              -> <Elem-CF>
  <<Elem-CF>-VAR>                                      -> <Elem-CF>
  [\110]                                               -> "n"
  [\42]                                                -> "*"
  [\40]                                                -> "("
  [\41]                                                -> ")"
  "x"                                                  -> <<Elem-CF>-VAR>
  "y"                                                  -> <<Elem-CF>-VAR>
  "z"                                                  -> <<Elem-CF>-VAR>
  [\120]                                               -> "x"
  [\121]                                               -> "y"
  [\122]                                               -> "z"
                                                       -> <LAYOUT?-CF>
  <LAYOUT-CF>                                          -> <LAYOUT?-CF>
```

Figure 7: Monoid in Kernel-SDF

To be able to make all this happen, the following issues must be addressed:

- CASL is a modular specification formalism with a powerful import mechanism, it allows parameterization of modules and renaming.
- Operator definitions can be global or local, and so can a module import.
- Variables may be defined local to the axioms or globally defined. Globally defined variables are visible for the subsequent axioms of the enclosing specification. Any subsequent declaration (global or local) of a variable with the same name overrides a previous one.
- Axioms, operator definitions and variable definitions can be intermingled, as long as operators and variables are defined before they are used.

The import mechanism necessitates inspecting all imported modules when parsing a module: all global definitions, which may have been affected by an imported module, must be retrieved.

Definitions of sorts, predicates, and operations involve formulae and terms; moreover, variables may be declared by explicit quantifiers within formulae (which can also occur within conditional terms). Because of this fact, each axiom, in principle, might require constructing a new parse table. The speed of parse table generation is therefore of significant practical relevance.

A parse table built to cope with syntax introduced by a particular axiom is constructed from the respective sets of visible sorts, operator definitions, and variables as they pertain to that particular axiom. The generation process consists of two steps. In the first step (the normalization step) the Kernel-SDF grammar is derived. In the second step the parse table is produced from the Kernel-SDF grammar. The first step, the grammar normalization step, is the most interesting from the current perspective, because it is at this point that the appropriate set of extra production rules must be added, in order to recognize layout as well as the overall structure of formulas. The complete set of context-free grammar rules that have to be added can be found in [11].

The Kernel-SDF syntax for the Monoid example is given in Figure 7. Note that this Figure does not reveal all details: for brevity, the Kernel-SDF definition of LAYOUT is omitted.

Using this Kernel-SDF specification, it is fairly straightforward to construct a parse table for the unparsed expressions. After a successful parse of such an unparsed expression, the obtained CASFIX representation must be substituted for the unparsed expression in the original abstract syntax tree, yielding a more detailed abstract syntax tree.

3.4 Drawbacks and Shortcomings

The architecture of the CASL parser presented in Section 1.1 is fairly complex at first glance. The need to translate parse trees, as produced by the SGLR parser, to CASFIX format is mainly responsible for this. One

could conceive of a parser implementation with a parameterized output format: such a parser could generate arbitrary representations of syntax trees in a flexible way. In the context of CASL, specifically, it could be instantiated to produce CASFIX directly.

However, the current SGLR implementation cannot (yet) boast of this feature: its output is the rich and versatile ASFIX format. Therefore, an translation from ASFIX to CASFIX must be performed at every stage. In this translation, some of the wealth of the ASFIX representation is lost: it carries only the abstract syntax minus layout. Note that comments are considered layout in the CASL language definition, and are consequently also discarded here.

4. REPRESENTING CASL TERMS USING ASF+SDF TECHNOLOGY

In order to make CASL a viable formalism for the specification of (complex) systems, a powerful suite of tools to create, manipulate, proof, typeset, and execute CASL specifications is needed. Reuse of existing tools is preferred over reinventing the wheel. However, existing tools are idiosyncratic, typically: they define their own interfaces and representation formats. This observation indicates a need for a common intermediate exchange format, that enables efficient integration of existing tools. Such an intermediate format should allow the storage by a tool of auxiliary information that may be invisible for others. To satisfy these requirements, the COFI community has decided to use ATERMS [8] as its exchange format.

As an alternative possibility, SGML has also been taken in account while considering exchange format options [30]. If this choice had to be redone at present, some other formats, among which abstract syntax definition language (ASDL) [36, 16] and eXtensible Markup Language (XML) [38] deserves explicit mention, would merit consideration.

ATERMS were originally developed to represent parse trees within the ASF+SDF Meta-Environment.

4.1 ATERMS

ATERMS is a generic formalism for the representation of structured information. It is both human-readable and easy to process automatically. A number of libraries that implement the functionality of creating and manipulating terms provide an API for the ATERMS formalism. The primary application area of ATERMS is the exchange of information between components of a programming environment, such as a parser, a (structure) editor, a compiler, and so on. The following data are typically represented as ATERMS: programs, specifications, parse tables, parse trees, abstract syntax trees, proofs, and the like. A generic storage mechanism, called *annotation*, accommodates associating extra information that may be of relevance somehow to specific ATERMS under consideration.

Examples of objects that are typically represented as ATERMS are:

- *constants*: `abc`.
- *numerals*: `123`.
- *literals*: `"abc"` or `"123"`.
- *lists*: `[]`, `[1, "abc", 3]`, or `[1, 2, [3, 2], 1]`.
- *functions*: `f("a")`, `g(1,[])`, or `h("1", f("2"), ["a","b"])`.
- *annotations*: `f("a")`{`[g,g(2,["a"])]`} or `"1"`{`[l,[1,2]],[s,"ab"]`}.

ATERMS can be qualified as an *open*, *simple*, *efficient*, *concise*, and *language independent* solution for the exchange of data structures between distributed applications.

ATERMS Libraries In order to employ ATERMS and to provide their associated operations, libraries of predefined functions are available. Currently, implementations in the programming languages C and JAVA exist.

Files that contain ATERMS can either be stored in a compact binary format or in a significantly more space-consuming textual format. In textual format, ATERMS can easily be processed by common off-the-shelf tools,

like LEX/YACC. Both the C and JAVA library implementations provide facilities to parse and unparse ATERMS. The C version also provides functions for mapping the textual form into the binary one and vice versa.

Furthermore, both library implementations ensure maximal sharing when creating and manipulating terms – unless this feature is explicitly disabled. Both implementations perform automatic garbage collection, thus freeing storage associated to terms that are no longer in use.

The ATERMS library is documented extensively in its user manual [22].

4.2 CASFIX

By using CASL-specific 'keywords' as ATERM AFuns, the abstract syntax as defined in the CASL language summary [11] can be represented in ATERMS in a straightforward manner.

It should be noted that there are many ways of defining the abstract syntax of CASL in terms of ATERMS. We will restrict ourselves to describing the approach decided upon by the COFI working group.

The selected approach is based on creating a unique ATERM construct for each abstract syntax rule. This results in a relatively large number of different AFuns, but has the benefit of making the representation of abstract syntax trees more compact.

The 'translation' of abstract rules into equivalent ATERM constructs is fairly simple, as the following translation rule illustrates:

```
SORT ::= "rule" MEMBER1 MEMBER2
⇒
rule(<MEMBER1>,<MEMBER2>)
```

There are several design issues to be addressed. The principal ones are how to deal with so-called 'chain rules', and how list structures are to be represented.

AFun Instantiation CASL-specific function names give rise to instantiating an AFun:

```
"basic-spec"  -> AFun
"forall"      -> AFun
"op-defn"     -> AFun
...
```

Function names are taken from the abbreviated abstract syntax definition in [11].

Additionally, specific AFuns are introduced to represent the anonymous abstract syntax rules (the chain rules), e.g.:

```
BASIC-ITEMS ::= SIG-ITEMS
```

An AFun is introduced e.g.:

```
"SIG-ITEMS" -> AFun
```

Finally, AFuns to represent lists are introduced, e.g.:

```
"BASIC-ITEMS*" -> AFun
```

Mapping Rules The mapping from the abstract syntax of CASL to an ATERMS equivalent is based on the type of the abstract syntax rule.

- Rules with only a terminal in their right-hand side:

  ```
  QUANTIFIER :: "forall"
  ⇒
  forall
  ```

  The terminal is mapped onto an ATERM consisting of a single AFun: `forall`.

- Rules consisting of a constructor and a list of simple nonterminals:

```
OP-ITEM ::= "op-defn" OP-NEMA OP-HEAD TERM
⇒
op-defn(<OP-NAME>,<OP-HEAD>,<TERM>)
```

  The abstract syntax rule is transformed into an ATERM consisting of the `AFun` `op-defn` with three argument ATERMS for `OP-NAME`, `OP-HEAD`, and `TERM`.

- Chain rules:

```
BASIC-ITEMS ::= SIG-ITEMS
⇒
SIG-ITEMS(<SIG-ITEMS>)
```

  The sort name on the right hand side is used as the `AFun`.

- Rules with a constructor and one or more lists:

```
BASIC-SPEC ::= "basic-spec" BASIC-ITEMS*
⇒
basic-spec(BASIC-ITEMS*([<BASIC-ITEMS>]))
```

  This ATERMS expression should be read as follows: `basic-spec` is an `AFun` that indicate that a node of type `basic-spec` is constructed. The `AFun` `BASIC-ITEMS*` indicates a node that contains a (possibly empty) list of `BASIC-ITEM`s. Square brackets delimit the actual list; list elements are separated by commas.

The full mapping can be found on the WWW[6].

4.3 Annotations

The term 'annotation' is overloaded. First of all, the ATERMS support annotations; in that context, it means that nodes in the abstract syntax tree are extended with additional information which is invisible, but can still be accessed by explicit demand. This annotation mechanism can be used by all kinds of tools to store extra information, e.g. a parser can store position information, or pretty printers can store font information.

Secondly, annotations can also occur within a CASL specification, see [11, 27, 32]. There exist annotations related to labels, displaying, parsing, and semantics.

Examples of syntactical annotations are `%left assoc`, `%right assoc`, and `%prec`; examples of semantical annotations are `%cons` and `%def`. For the exact meaning of these annotations we refer to [32]. The `%cons`, `%def`, `%left assoc`, and `%right assoc` annotations can only occur at a restricted number of positions in a CASL specification.

Some annotations, such as the `%prec`, may occur in arbitrary places in a CASL specification. The CASL language summary [11] is not very specific about where the various types of annotations may occur. If these locations are fixed, the concrete syntax of CASL can be adapted in order to deal with these annotations, however, it is also possible to consider annotations as a kind of layout. The latter approach would prevent the need to adapt the CASL syntax every time a new annotation is introduced.

Annotations occurring in well-defined locations in CASL specification must also be encoded in the CASFIX representation at the appropriate nodes, with the correct name, as generally approved upon.

In order to prevent excessive use of annotations both at the CASL syntax level as well as at the tool level, annotations have to be approved by the COFI community. Naturally, it is allowed for a tool to add an 'internal' annotation temporarily, but such an annotation may not be exported to the outside world.

[6] `http://www.cwi.nl/~markvdb/cofi/casl.html`

5. CONCLUSIONS

Generic language technology provided by ASF+SDF proved to be helpful for prototyping the concrete syntax of CASL. The straightforward mapping from EBNF to SDF enabled us to interact directly with the concrete syntax definition of CASL. Serious lexical and syntactical ambiguities were detected at an early stage. This flexibility is based on the SGLR parsing technology which removes the need for (complex) grammar transformations.

Although ASF+SDF proved to be a useful vehicle for prototyping the concrete syntax of CASL, a number of shortcomings can be identified. Firstly, the user-defined syntax of CASL proved to be challenging for ASF+SDF; see [37] for details. The CASL syntax proved to be an interesting test case for the development of both SGLR parser and the parser generator: the complexity of the CASL syntax continuously pushed the envelope, putting a heavy burden on ambiguity-handling mechanisms. As larger parts of the syntax were completed, and more complex examples were held against the current state of technology, it was rapidly adapted, tweaked, corrected, and revised.

Secondly, although the CASL syntax is described entirely in SDF and specifications can be parsed inside the Meta-Environment, the existence of a straightforward translation to other parser generator formalisms like LEX/YACC [26, 21], JAVACC, etc., is not guaranteed, mainly because of the limitation of the latter to the class of LALR grammars. Using the powerful and efficient parsers provided by current ASF+SDF technology, the need to migrate from SDF to LEX/YACC-like formalisms becomes less pressing.

The development of parsers for CASL and the definition of an exchange format formed the first steps on the long path that lead to the creation of new tools and the adaptation of existing ones. Although the current ASF+SDF Meta-Environment is entirely based on ATERMS, it is still a question whether ATERMS are indeed powerful enough to contain all information needed by any conceivable CASL-tool – past, present and future. However, previous experience, including experiments carried out in Bremen [28], Nancy [23], and Edinburgh [2], suggest a positive answer; future experiments, with more, and more complex, tools built upon ATERMS foundations, will help gain a firmer grasp on this subject.

ACKNOWLEDGEMENTS

We thank Jan Heering and Pieter Olivier for reading the draft versions of this paper.

References

1. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.
2. D. Baillie. Proving Theorems about Algebraic Specifications. Master's thesis, University of Edinburgh, Department of Computer Science, 1999.
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 – a software coordination architecture. *Science of Computer Programming*, 31(2-3):205–229, July 1998.
5. M. Bidoit and C. Choppy. ASSPEGIQUE: an integrated environment for algebraic specifications. In H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, editors, *Formal Methods and Software Development - Proceedings of the International Joint Conference on Theory and Practice of Software Development 2*, volume 186 of *LNCS*, pages 246–260. Springer-Verlag, 1985.
6. P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In José Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1996.
7. P. Boullier. Contribution à la construction automatique d'analyseurs lexicographiques et syntaxiques. Thèse d'Etat, Université d'Orléans, 1984.
8. M.G.J. van den Brand, H.A. de Jong, P. Klint, and P. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30:259–291, 2000.
9. M.G.J. van den Brand, T. Kuipers, L. Moonen, and P. Olivier. Design and implementation of a new asf+sdf meta-environment. In A. Sellink, editor, *Proceedings of the Second International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)*, Workshops in Computing, Amsterdam, 1997. Springer/British Computer Society.
10. CoFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents accessible by HTTP[7] and FTP[8], 1998.
11. CoFI-LD. CASL – The CoFI Algebraic Specification Language – Summary, version 1.0. Documents/CASL/Summary-v1.0, in [10], 1998.

[7] `http://www.brics.dk/Projects/CoFI/`
[8] `ftp://ftp.brics.dk/Projects/CoFI/`

12. A. van Deursen. *Executable Language Definitions: Case Studies and Origin Tracking Techniques*. PhD thesis, University of Amsterdam, 1994.
13. A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.
14. A. van Deursen and P.D. Mosses. Executing Action Semantics descriptions using ASF+SDF. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Algebraic Methodology and Software Technology (AMAST'93)*, Workshops in Computing, pages 415–416. Springer-Verlag, 1993. System demonstration.
15. J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.
16. D.R. Hanson. Early Experience with ASDL in lcc. *Software—Practice and Experience*, 29(3):417–435, 1999.
17. 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.
18. J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *IEEE Transactions on Software Engineering*, SE-16:1344–1351, 1990.
19. J. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Language, and Computation*. Addison-Wesley, 1979.
20. D. Hutter and C. Sengler. INKA, The Next Generation. In M.A. McRobby and J.K. Slaney, editors, *Automated Deduction – CADE-13*, volume 1104 of *LNAI*, pages 288–292. Springer, 1996.
21. S.C. Johnson. *YACC: yet another compiler-compiler*. Bell Laboratories, 1986. UNIX Programmer's Supplementary Documents, Volume 1 (PS1).
22. H.A. de Jong and P. Olivier. ATerm Library User Manual, 1999. Available via HTTP[9].
23. H. Kirchner and C. Ringeissen. Executing CASL Equational Specifications with the ELAN Rewrite Engine. Note T-9, in [10], 1999.
24. P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.
25. P. Klint and E. Visser. Using filters for the disambiguation of context-free grammars. In G. Pighizzini and P. San Pietro, editors, *Proceedings ASMICS Workshop on Parsing Theory*, pages 1–20, 1994. Published as Technical Report 126–1994, Computer Science Department, University of Milan.
26. M.E. Lesk and E. Schmidt. *LEX - A lexical analyzer generator*. Bell Laboratories, unix programmer's supplementary documents, volume 1 (ps1) edition, 1986.
27. T. Mossakowski. Standard annotations for parsers and static semantic checkers — a proposal. Note T-6, in [10], 1998.
28. T. Mossakowski. CASL: From Semantics to Tools. In S. Graf and M. Schwartzbach, editors, *TACAS'2000*, volume 1785 of *LNCS*, pages 93–108. Springer-Verlag, 2000.
29. P. D. Mosses. *Action Semantics*. Cambridge University Press, 1992.
30. P.D. Mosses. Potential use of SGML for the CASL interchange format. Note T-4, in [10], 1997.
31. J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.
32. M. Roggenbach and T. Mossakowski. Proposal of Some Annotations and Literal Syntax in CASL. Note L-11, in [10], 1999.
33. M. Tomita. *Efficient Parsing for Natural Languages*. Kluwer Academic Publishers, 1985.
34. E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.
35. F. Voisin. CIGALE: a tool for interactive grammar construction and expression parsing. *Science of Com-*

[9] `http://www.wins.uva.nl/pub/programming-research/software/aterm/`

puter Programming, 7(1):61–86, 1986.

36. D.C. Wang, A.W. Appel, J.L. Korn, and C.S. Serra. The Zephyr Abstract Syntax Description Language. In *Proceedings of the Conference on Domain-Specific Languages*, pages 213–227, 1997.
37. B. Wedemeijer. Introduction and Basic Tooling for Casl using ASF+SDF. Technical Report P9809, University of Amsterdam, Programming Research Group, 1998.
38. Extensible markup language (XML) 1.0. Technical report, World Wide Web Consortium, 1998. Available at: `http:www.w3.org/TR/REC-xml`.