ASF+SDF: a functional language to prototype (programming) languages Extended abstract

M.G.J. van den Brand[1]

1: Technical University Eindhoven
Department of Mathematics and Computer Science
Den Dolech 2
NL-5612 AZ Eindhoven
The Netherlands
m.g.j.v.d.brand@tue.nl

Résumé

Although ASF+SDF stands for Algebraic Specification Formalism plus Syntax Definition Formalism, the formalism has more a functional flavor than an algebraic flavor. The main purpose of this formalism is to describe both the syntax and the semantics of (programming) languages. Initially, ASF+SDF was developed to prototype languages, among others domain specific languages, in the last decade the application area shifted towards software renovation. ASF+SDF is used for the restructuring of COBOL code, among others, GOTO elimination, subroutine introduction, and data migration. The ASF+SDF formalism is supported by an integrated development environment, the ASF+SDF Meta-Environment. Various components of this environment are developed using ASF+SDF itself and compiled to C code.

1. Introduction

The focus of research in the field of generic language technology is on the development of fundamental techniques for (programming) language processing: analysis, transformation, and compilation. Besides the development of formalisms for describing the syntax and semantics of programming languages, tools for processing languages and programs are developed as well. The formalism ASF+SDF [2] [11] and the corresponding integrated development environment, the ASF+SDF Meta-Environment [15][3] are examples of results obtained in this field of research. The scope of research with respect to ASF+SDF and the Meta-Environment is on exploring new fundamental concepts, such as declarative description of (programming) languages, incremental generation techniques, efficient term rewriting engines, advanced parsing technology, and new analysis techniques.

The ASF+SDF formalism [2] [11] is a formalism for the definition of syntactic and semantic features of (programming) languages, but it can also be used for the formal specification of a wide variety of software engineering problems.

This paper gives an overview of a number of applications of ASF+SDF. We give a brief introduction to ASF+SDF and the Meta-Environment. Finally we draw some conclusions on the applicability of ASF+SDF with respect to certain applications.

2. Asf+Sdf

Asf+Sdf is a general-purpose, executable, algebraic specification formalism. Its main application areas are the definition of the syntax and the static semantics of (programming) languages, program transformations and analysis, and for defining translations between languages. Asf+Sdf provides

- general-purpose algebraic specification formalism based on (conditional) term rewriting.
- modular structuring of specifications.
- integrated definition of lexical, context-free, and abstract syntax.
- user-defined syntax, allowing you to write specifications using your own notation.
- traversal functions (for writing very concise program transformations), memo functions (for caching repeated computations), list matching, and more.

The Asf+Sdf formalism is a combination of two formalisms: Asf (the Algebraic Specification Formalism [2, 11]) and Sdf (the Syntax Definition Formalism [13]). Sdf is used to define the concrete syntax of a language, whereas Asf is used to define conditional rewrite rules; the combination Asf+Sdf allows the syntax defined in the Sdf part of a specification to be used in the Asf part, thus supporting the use of user-defined syntax when writing Asf equations. Asf+Sdf also allows specifications to be split up into named modules, enabling reuse.

2.1. Syntax Definition Formalism

Sdf is a declarative formalism used to define the concrete syntax of languages: not only programming languages, for example Java and COBOL, Sdf can also be used to define specification languages, such as Chi, Elan, and Action Semantics. Sdf does not impose any restrictions on the class of grammars used, it accepts arbitrary, cycle-free, context-free grammars, which may even be ambiguous. Since the class of all context-free grammars is closed under union, a modular definition of grammars is possible in Sdf, unlike other (E)BNF formalisms.

Although the full power of arbitrary context-free grammars is hardly necessary when defining the syntax of a programming language (except for languages like COBOL, PL/I), modularity is essential for reuse of specific language constructs in various language definitions. See Figure 1 for an example of an Sdf module.

2.2. Algebraic Specification Formalism

Asf is a declarative formalism used to define the semantics of (programming) languages. In a way it can be considered as a first-order functional programming language. It provides conditional equations, also allowing negative conditions. The concrete syntax defined in the corresponding Sdf module and in the transitive closure of any imported modules (only the exported sections, of course) can be used when writing the conditional equations of an Asf module. Traversal functions [7] provide a concise way of defining an Asf function which traverse the term and perform transformation and/or accumulation operations on specific nodes in the underlying term without providing all intermediate rewrite steps. See Figure 2 for an example of an Asf module.

3. ASF+SDF Meta-Environment

The development of Asf+Sdf specifications is supported by an interactive integrated programming environment, the Meta-Environment [15][3]. This programming environment provides syntax directed

```
module basic/Booleans

imports basic/BoolCon
exports
  sorts Boolean

  context-free syntax
    BoolCon               -> Boolean
    Boolean "|" Boolean   -> Boolean {left}
    Boolean "&" Boolean   -> Boolean {left}
    "not" "(" Boolean ")" -> Boolean
    "(" Boolean ")"       -> Boolean {bracket}

  context-free priorities
    Boolean "&" Boolean -> Boolean >
    Boolean "|" Boolean -> Boolean

hiddens
  context-free start-symbols Boolean

  imports basic/Comments
  variables
    "Bool" -> Boolean
```

Figure 1: The SDF module of the Boolean language

```
equations

  [B1]   true | Bool   = true
  [B2]   false | Bool  = Bool

  [B3]   true & Bool   = Bool
  [B4]   false & Bool  = false

  [B5]   not ( false ) = true
  [B6]   not ( true )  = false
```

Figure 2: The ASF module of the Boolean language

editing facilities for both the SDF and ASF parts of modules as well as for terms, well-formedness checking of modules, interactive debugging of ASF equations, and visualisation facilities of the import graph and parse trees. The Meta-Environment provides

- interactive support for writing a formal specification of a problem.
- an interactive environment for a new (application) language.
- support for analyzing or transforming programs in existing languages.

Besides the basic functionality, like editing, rewriting, debugging, and checking, the Meta-Environment offers integrated access to predefined modules containing

- a collection of grammars of programming and specification languages, such as Java, C, BOX, and SDF itself.
- basic data types such as `Booleans`, `Naturals`, and `Strings`.
- basic data structures, such as `Sets`, `Tables`, the basic data structures are parameterized.
- BOX operators to guide the formatting of text in a declarative manner.
- a data structure to manipulate warnings and error messages.
- functionality to access the underlying position information of subterms.

The user interface of the Meta-Environment is shown in Figure 3. The figure shows the modules of Sdf-Checker. The left pane shows a tree-structured view of the modules, and the right pane shows the graph module with import relations.

4. Applications

The obvious application areas for ASF+SDF and the Meta-Environment technology are the design and implementation of domain specific languages, software renovation, and advanced code generators. In Section 4.1 we discuss the use of ASF+SDF within the Meta-Environment. In 4.2 we discuss a few of the most important academic applications and in Section 4.3 we will discuss a number of industrial applications.

4.1. ASF+SDF specific applications

The core business of the Meta-Environment is language processing. ASF+SDF is suited to be used as an algebraic specification formalism for specifying language processing tools. So, it is logical to use ASF+SDF to implement the following components of the Meta-Environment:

- the ASF2C compiler [5];
- BOX toolset [9] [8];
- SDF normalizer as part of the parsetable generator for SDF [19];
- SDF well-formedness checker.

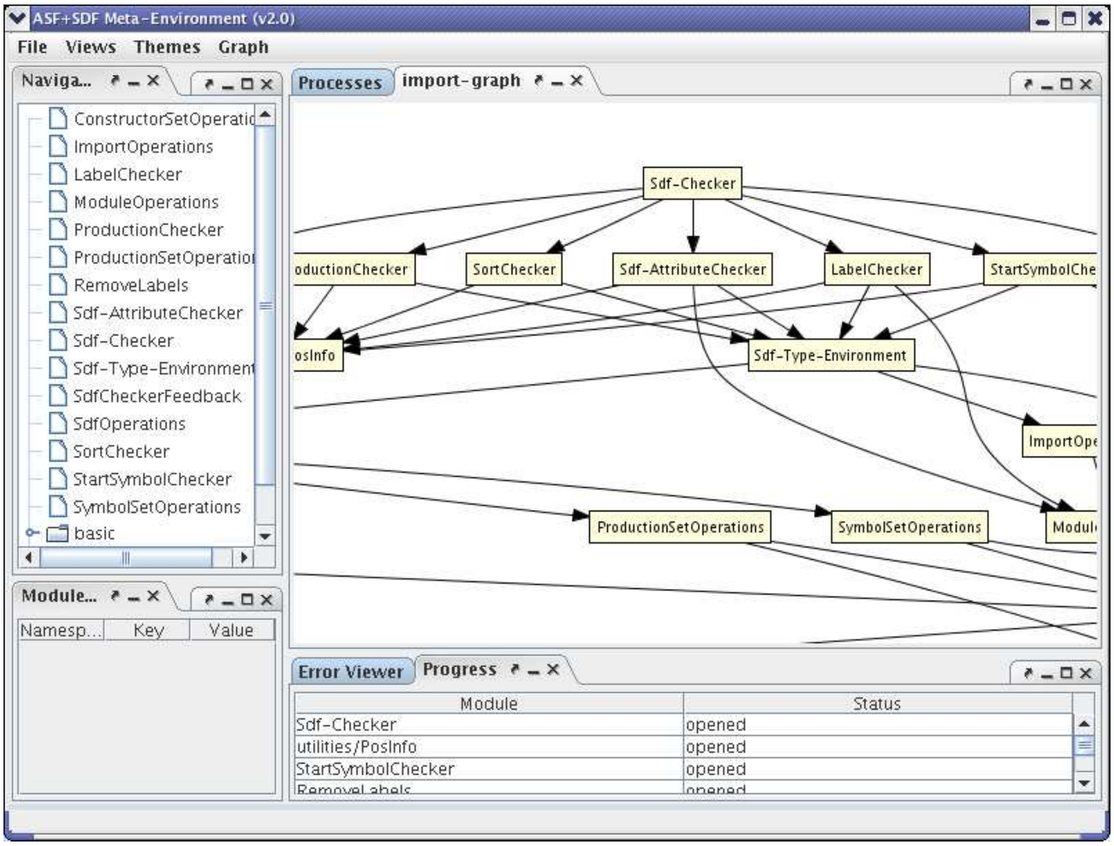

Figure 3: The user-interface of the Meta-Environment is a browser that provides a graphical and a textual view of the modules

The Asf2C compiler [5] compiles Asf+Sdf specifications to efficient C code. Every Sdf function (with Asf equations) is translated to a C function that contains an optimized matching automaton for the left-hand sides of the equations and conditions. The Asf+Sdf functions in conditions and the right-hand side of a matched equation are translated to direct function calls.

The Box toolset [9] [8] provides a fully integrated way of defining the unparsing of terms manipulated via Asf+Sdf. It consists of the Box formalism and a Box interpreter (Pandora) for translating Box expressions to either Ascii text, HTML code, or LaTeX. The unparsing of language constructs is defined using plain Asf equations.

The Sdf normalizer is described in Visser's PhD thesis [19], see Chapters 6 through 10[1].

Sdf well-formedness checker Sdf is in fact a collection of syntax definition formalisms. The core of Sdf is formed by kernel-Sdf, which provides normalized syntax rules. On the kernel level there is, for instance, no distinction between lexical and context-free syntax and the modular structure has been resolved. An Sdf specification is *normalized* to this kernel-Sdf, by performing grammar transformations.

An Sdf definition used in combination with Asf does not support kernel syntax constructions. Furthermore, in order to parse the Asf equations, the Sdf specification is "extended" with a module

[1]http://www.cs.uu.nl/people/visser/ftp/Vis97.ps.gz

containing the syntax rules for the equations. In order to prevent "clashes" the nonterminals used in this ASF module are not allowed in an arbitrary SDF definition. Some other restrictions are imposed on SDF constructs in order to able to rewrite the parsed terms via ASF, for example, the separator in lists should always be a literal (`{Bool Bool}+` is not a valid SDF symbol in combination with ASF, whereas the list construct `{Statement ","}*` is a valid symbol). These requirements are not checked during normalization nor parsetable generation. The reason for this is that the normalization and parsetable generation support the largest class of SDF. Therefore a separate SDF well-formedness checker has been implemented.

In order to support an efficient development of SDF definitions, the Meta-Environment provides an SDF-checker, which checks a number of well-formedness conditions. The most important checks are

- whether no kernel syntax constructions are used.
- whether nonterminals are used which are part of the "ASF" language.
- whether all used nonterminals (sorts in SDF terminology) are defined in some right-hand side of a production rule.
- whether at least one start symbol is defined.
- whether the traversal functions have the correct combination of attributes.

The result of running this checker is a list of warnings and errors based on the predefined error module in the library. The generated messages contain position information to connect the error to the exact location in a module where the error occurred. This specification uses traversal functions in order to obtain information from all parts of an SDF definition. Figure 4 gives the essential rule for finding nonterminals which are in fact used within the ASF syntax. The function `get-location($Sort)` obtains the position information for the sort `$Sort`. This `get-location` function is a function defined in the module `utilities/PosInfo`.

4.2. Academic applications

The academic applications of ASF+SDF are mainly in the field of programming language prototyping, transformation, and compilation. In this section we consider applications which are not directly related to ASF+SDF itself. Three projects in the area of language prototyping are

- the prototyping of the next generation of the action semantic formalism [12] [14]. Besides the prototyping of this formalism an environment for this formalism is developed [6].
- the prototyping of RSCRIPT formalism and tooling. RSCRIPT provides a relational approach to software analysis [16].
- the prototyping of formalism Chi [1] at the Mechanical Engineering Group at the Technical University of Eindhoven. The purpose of the Chi formalism is the specification of the dynamics and control of production plants and mechanical modelling in order to perform calculations on the performance of these production plants.

Projects in the area of program transformation and compilation are as follows:

- the validation of distributed algorithms with a rewriting kernel dedicated to TLA+ specifications [17] at IRIT (University of Toulouse). This project mainly uses traversal functions in order to describe the transformations in a very concise way. Furthermore our BOX pretty printing technology is used to regenerate parseable TLA+ specifications again.

```
...
imports
   utilities/PosInfo[Sort]
...
exports
  context-free syntax
    check-asf-sorts(Sort, {Error ","}*)
      -> {Error ","}* {traversal(accu, break, top-down)}
  ...
hiddens
  variables
    "$Msgs"[0-9]*     -> {Error ","}*
    "$Sort"[0-9]*     -> Sort
    "$String"[0-9]*   -> StrCon
    "$Location"[0-9]* -> Location

equations
  ...
  [] is-asf-sort($Sort) == true,
     $Location := get-location($Sort),
     $String := symbol2str($Sort)
     ====>
     check-asf-sorts($Sort, $Msgs) =
       $Msgs, make-error("Usage of asf equation sort is not allowed ",
                         $String, $Location)
  ...
```

Figure 4: The SDF function and ASF equation for finding ASF nonterminals in SDF

- the implementation of a compiler for the formalism Chi [1]. Chi programs are translated to either C or Python.
- the migration of legacy databases to relational databases together with the adaptation of the corresponding program code [10].

4.3. Industrial applications

There are three main industrial applications areas of ASF+SDF which are very similar to the academic application areas: prototyping of domain specific languages (DSLs), software renovation, and code generation. In this section we discuss the software renovation activities in more detail. The prototyping of DSLs and code generation is discussed in [4].

Various projects in the field of software renovation, such as reverse engineering and re-engineering have been carried out in cooperation with industrial partners since 1998. The powerful generalized parsing technology allowed us to tackle both the problem of handling various dialects of Cobol as well as the problem of embedded languages in Cobol, such as SQL, assembler, and CICS.

In various software renovation projects ASF+SDF has been applied to restructure Cobol programs, see [18]. The main goal of this work was to improve maintainability of the code. The restructuring consisted of a number of steps, among others

- the introduction of scope terminators, such as `END-IF`;
- the removal of as many GOTOs as possible;
- the introduction of subroutines by means of `PERFORM` statements;

- the introduction of loops by means of inline `PERFORM` statements;
- the prettyprinting of the resulting Cobol program.

5. Conclusions

The application areas of ASF+SDF and Meta-Environment are very diverse. However the unifying factor is language processing. The shift from prototyping small languages, DSLs, to software renovation has had a tremendous effect on the underlying technology. It triggered the development of scalable language processing technology. The Meta-Environment was completely redesigned using component-based software development technology. The focus shifted from incremental techniques to scalability, flexibility, re-usability and efficiency of tools. This development not only opened new application areas, but also enabled us to promote and distribute the underlying technology to other research groups.

Obtaining the ASF+SDF Meta-Environment

The Meta-Environment can be downloaded from: `http://www.cwi.nl/projects/MetaEnv/`.

Acknowledgements

I would like to thank all current and former members of the Generic Language Technology group at CWI for making the Meta-Environment work. Furthermore I would like to thank all people who have used and still use ASF+SDF and the Meta-Environment, either to do research or to apply it to solve (complex) problems.

Bibliographie

[1] D.A. van Beek, K.L. Man, M.A. Reniers, J.E. Rooda, and R.R.H. Schiffelers. Syntax and consistent equation semantics of hybrid Chi. *Journal of Logic and Algebraic Programming*, 2005. To appear.

[2] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press/Addison-Wesley, 1989.

[3] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *CC'01*, volume 2027 of *LNCS*, pages 365–370. Springer-Verlag, 2001.

[4] M.G.J. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E.A, van den Meulen. Industrial applications of ASF+SDF. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of *LNCS*. Springer-Verlag, 1996.

[5] M.G.J. van den Brand, J. Heering, P. Klint, and P.A. Olivier. Compiling language definitions: The asf+sdf compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.

[6] M.G.J. van den Brand, J. Iverson, and P.D. Mosses. An Action Environment. *Science of Computer Programming*, 2005. to appear.

[7] M.G.J. van den Brand, P. Klint, and J.J. Vinju. Term rewriting with traversal functions. *ACM Transactions on Software Engineering and Methodology*, 12(2):152–190, 2003.

[8] M.G.J. van den Brand, A.T. Kooiker, N.P. Veerman, and J.J. Vinju. An industrial application of context-sensitive formatting. Technical Report SEN-R0510, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 2005.

[9] 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.

[10] A. Cleve, J. Henrard, and J-L. Hainaut. Co-transformations in information system reengineering. In *Second International Workshop on Meta-Models, Schemas and Grammars for Reverse Engineering (ATEM'04)*, volume 137-3 of *ENTCS*, pages 5–15, 2004.

[11] 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.

[12] K.-G. Doh and P.D. Mosses. Composing programming languages by combining action-semantics modules. In M.G.J. van den Brand and D. Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier, 2001.

[13] 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.

[14] J. Iverson and P.D. Mosses. Constructive Action Semantics for Core ML. *IEE Proceedings — Software*, 152(2):79–98, 2005.

[15] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.

[16] P. Klint. *A Tutorial Introduction to RScript — a Relational Approach to Software Analysis*, 2005. http://homepages.cwi.nl/ paulk/publications/rscript-tutorial.pdf.

[17] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[18] N. Veerman. Revitalizing modifiability of legacy assets. *Software Maintenance and Evolution: Research and Practice, Special issue on CSMR 2003*, 16(4–5):219–254, 2004.

[19] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.