"Generating Tools from Algebraic Specifications"
Abstracts of the papers from the ASF+SDF'95 Workshop
CWI, Amsterdam, May 11 & 12, 1995

[1] J. Heering and P. Klint. The prehistory of ASF+SDF (1980-1984). pages 1–4.

[2] P. Klint. The evolution of implementation techniques in the ASF+SDF meta-environment. pages 5–26.

Abstract: The ASF+SDF Meta-environment is an interactive development environment for formal language definitions. It is both a *meta-environment* supporting fully interactive editing of modular language definitions written in the formalism ASF+SDF and a *generator* for dedicated environments for defined languages.

The actual development of this system started in 1985 as part of the GIPE (Generation of Interactive Programming Environments) projects [HKKL86]. Now, ten years later, it is worthwhile to assess what has been achieved and, more importantly, which problems are still to be addressed.

A historical and at times methodological perspective is necessary in such an assessment. However, rather than evaluating *all* aspects of the system I will concentrate on the evolution of the implementation techniques being used. This implies that I will neither assess the formalism ASF+SDF itself nor discuss more fundamental research questions related to topics like modularization, higher-order formalisms, compiler generation, and the like.

Instead, I will try to distill lessons from the patterns that can be identified in the evolution of the implementation techniques used so far. These lessons are used to make projections for the future.

The ASF+SDF Meta-environment has been developed as part of the Centaur system, the end result of the GIPE projects. The analysis to be given here will completely focus on the Meta-environment and will only discuss those aspects of Centaur that are directly relevant to this analysis. Other aspects of Centaur will be mostly ignored.

The paper gives a chronological account: past, present, and future. Readers interested in *pre* historic considerations should consult [1]. An overview of the Meta-environment is given in [Kli93].

[3] J. Heering. An algebraic machine for imperative programs (abstract). page 27.

Abstract: In [Fie92] Field introduced PIM, an algebraic machine for the analysis, transformation, and optimization of imperative programs. For these purposes the algebraic framework is particularly attractive since it provides a uniform setting for modelling the standard dynamic semantics of a language (implemented as a confluent term rewriting system) as well as arbitrary semantics preserving program transformation and partial evaluation (using equations of the complete logic).

Whereas the standard semantics involves only rewriting of PIM ground terms, program transformation and partial evaluation amount to rewriting of PIM terms containing (meta)variables. The appropriate notion of completeness in this setting is not the well-known completeness of equational logic, but ω-completeness [Hee86, BH94] of the specification of PIM. Making an algebraic specification ω-complete usually requires hard work (if it is possible at all), and PIM is no exception.

[4] M. G. J. van den Brand, S. M. Eijkelkamp, D. K. A. Geluk, H. Meijer, H. R. Osborne, and M. J. F. Polling. Program transformations using ASF+SDF. pages 29–52.

Abstract: This paper describes the ongoing work in the design and implementation of a program transformation system for educational purposes. The system is based on ASF+SDF for the implementation of the application of transformation rules and on SEAL for the implementation of the user interface. The transformation rules are given in a mixture of CIP-L and CLEAN: DISCIPLEAN. The DISCIPLEAN transformation rules are not ‘hard-wired’ in the system but they are stored in a kind of data base and they are available via an editor. This allows a flexible way of extending the set of transformation rules.

[5] S. Mauw and J.C. Mulder. A PSF library of data types. pages 53–64.

Abstract: We present a library of basic data types for the process specification formalism PSF. The specification is written in the algebraic specification formalism ASF, which is a sublanguage of PSF.

[6] S. Vigna. Specifying IMP(G) using ASF+SDF: A case study. pages 65–88.

Abstract: **Imp**(G) is a language based on distributive categories which allows to define uniformly classes of computable function arising from a basic set. The syntax of **Imp**(G) has been defined mathematically in terms of abstract syntax, so that any implementation has to face with a number of choices, ambiguities and polymorphisms of the categorical notation. In this paper we summarize the goals attained using ASF+SDF without resorting to hybrid functions. In particular, we describe how type checking was implemented equationally using the backtracking mechanism of list unification in order to obtain indifference to the associativity isomorphisms.

[7] E. Visser. A family of syntax definition formalisms. pages 89–126.

Abstract: In this paper we design a syntax definition formalism as a family of formalisms. Starting with a small kernel, various features for syntax definition are designed orthogonally to each other. This provides a framework for constructing new formalisms by adapting and extending old ones. The formalism is developed with the

algebraic specification formalism ASF+SDF. It provides the following features: lexical and context-free syntax, variables, disambiguation by priorities, regular expressions, character classes and modular definitions. New are the uniform treatment of lexical syntax, context-free syntax and variables, the treatment of regular expressions by normalization yielding abstract syntax without auxiliary sorts, regular expressions as result of productions and modules with hidden imports and renamings.

[8] M. V. Ferro and M.A.A. Pardo. Friendly incremental prototyping. pages 127–142.

Abstract: A development environment for interactive systems devoted to generate formal languages is described. Our system is organized around three cooperative modules. The first is a generator of general context-free parsers, following different parsing schemes among several available. The second is a generic incremental parsing facility that can be used to make the overall parsing process efficient in the context of program development. The third is an extensible user graphical interface that provides a complete set of customization and trace facilities for the system.

The final tool has been baptized ICE, after **I**ncremental **C**ontext-Free **E**nvironment. All components in ICE show a reasonable grade of efficiency both, in space and time. In an empirical comparison, it appears to be superior to other general context-free parsing environments and is comparable to the classic deterministic ones, when the context is not ambiguous. The cooperative architecture allows the modification of the environment, by adding or redesigning modules, with low impact on other components.

[9] S. Üsküdarlı. Specifying visual syntax. pages 143–154.

Abstract: This paper discusses visual syntax definition in a framework of generating programming environments for visual languages. In particular, the work is set in the context of the ASF+SDF Meta-environment [Kli93]. For the generation of a visual programming environment we need to be able specify visual languages. This paper discusses how we can define visual syntax for existing languages, and then outlines an approach to defining a visual specification formalism.

The basic idea is that given a language specification, such as *sets*, and a mapping from this language to a graphical syntax (using a visual definition language), we obtain a visual language definition from which we can generate visual tools. This approach can be exploited to define tools for a visual specification formalism yielding a *visual* environment for specifying visual languages.

[10] M. G. J. van den Brand. Pretty printing in the ASF+SDF meta-environment: Past, present, and future. pages 155–174.

Abstract: This paper gives an overview of the various pretty printing tools available in the ASF+SDF Meta-environment. The default pretty printer in the Meta-environment is probably the first component which has been fully specified in

ASF+SDF and compiled to C using the ASF2C compiler [KW93]. Some of the achievements concerning pretty printing will be discussed in this paper. A pretty printer generator and its use will be discussed, as well as the tuning of generated pretty printers, this will be demonstrated by means of a small example.

[11] S. Mauw and E. van der Meulen. Specification of tools for message sequence charts. pages 175–210.

Abstract: The recent formalization of the semantics of Message Sequence Charts enables the derivation of tools for MSCs directly from this formal definition. We use the ASF+SDF meta-environment to make a straightforward implementation of tools for transformation, simulation and requirements testing. In this paper we present the complete specification of the tools.

[12] A. van Deursen. A simple pi-calculus manipulation tool. pages 211–233.

Abstract: The π-calculus is a challenge for tool generation software, as it is subject to frequent changes. A specification of desirable functionality of tools for manipulating π-calculus expressions is given, and its extensibility is studied. The difficulties encountered during specification are discussed, as well as several proposals for easier tool specification.

[13] D. Naidich and T.B. Dinesh. Specifying an automated induction proof procedure in ASF+SDF. pages 233–254.

Abstract: We describe the implementation of an automated theorem prover for algebraic specifications, in an algebraic specification setting using the ASF+SDF Meta-environment. The current implementation is based on the implicit induction approach implemented in SPIKE [BKR92, BR93]. We consider the implementation as a case study on the tool generation within ASF+SDF which also provides an experimental basis for the research on automated induction. We consider the issues of the simplification strategies of implicit induction, the user interaction/heuristics in implicit induction, the computational effectiveness of ASF+SDF specifications, and the plausible improvements of the ASF+SDF specification language.

[14] T. B. Dinesh. Injection misdemeanors. pages 255–270.

Abstract: ASF+SDF allows injections in the presence of over-loading. The injections are generally perceived as a very useful feature of ASF+SDF. They have, however, troubled many – from implementors to specifiers. We elaborate on some of the problems (misunderstandings) encountered.

[15] J. F. Th. Kamperman and H. R. Walters. The new ASF compiler — An exercise in self-applicability. pages 271–293.

Abstract: We call a compiler *bootstrapped* when it is able to compile its own source. To achieve this situation, the compiler must be expressed in its own language, and an initial interpreter or compiler for this language must be available.

We describe the tools and techniques used for the bootstrapping of the new ASF compiler, which is itself expressed in ASF+SDF. As initial compiler, the previous version of the ASF compiler, ASF2C, and the meta-environment are used.

Both as a case-study in the manipulation of algebraic specifications, and as case-study in software-engineering of a portable compiler, the project is interesting to the ASF+SDF community.

References

[BH94] J. A. Bergstra and J. Heering. Which data types have ω-complete initial algebra specifications? *Theoretical Computer Science*, 124:149–168, 1994.

[BKR92] A. Bouhoula, E. Kounalis, and M. Rusinowitch. Automated mathematical induction. Technical Report 1636, INRIA, 1992.

[BR93] A. Bouhoula and M. Rusinowitch. Automatic case analysis in proof by induction. In *Proc. 13th IJCAI*, volume 1, pages 88–94, Chambery, France, 1993.

[Fie92] John Field. A simple rewriting semantics for realistic imperative programs and its application to program analysis. In *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 98–107, San Francisco, June 1992. Published as Yale University Technical Report YALEU/DCS/RR–909.

[Hee86] J. Heering. Partial evaluation and ω-completeness of algebraic specifications. *Theoretical Computer Science*, 43:149–167, 1986.

[HKKL86] J. Heering, G. Kahn, P. Klint, and B. Lang. Generation of interactive programming environments. In *ESPRIT '85: Status Report of Continuing Work*, pages 467–477. North-Holland, 1986. Part I.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering Methodology*, 2(2):176–201, 1993.

[KW93] J. F. Th. Kamperman and H.R. Walters. ARM, abstract rewriting machine. Technical Report CS-9330, Centrum voor Wiskunde en Informatica, 1993. Available by *ftp* from ftp.cwi.nl:/pub/gipe as KW93.ps.Z.