Centrum voor Wiskunde en Informatica

REPORT RAPPORT

Industrial applications of ASF+SDF

M. van den Brand, A. van Deursen, P. Klint, A.S. Klusener and
E. van der Meulen

Computer Science/Department of Software Technology

CS-R9622 1996

Report CS-R9622
ISSN 0169-118X

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

Industrial Applications of ASF+SDF

Mark van den Brand,[1] Arie van Deursen,[2] Paul Klint,[1,2]
Steven Klusener,[2,4] and Emma van der Meulen[3]

[1] University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam
[2] CWI, P.O. Box 94079, 1090 GB Amsterdam
[3] MeesPierson, Rokin 55, Amsterdam
[4] CAP Volmac, P.O. Box 2575, 3500 GN Utrecht

Email: arie@cwi.nl; URL: http://www.cwi.nl/∼arie/

Abstract

In recent years, a number of Dutch companies have used the algebraic specification formalism ASF+SDF. Bank MeesPierson has specified a language for describing interest rate products, their translation into COBOL, and their generation from interactive questionnaires. A consultancy company has specified a language to represent the company's object-oriented models, and the compilation of this language into Access. Bank ABN-AMRO has started investigating the use of algebraic specifications for renovating legacy COBOL systems. We discuss the implications of such projects for teaching algebraic specifications and software engineering, and the role students have been playing in these projects.

CR Subject Classification (1991): D.2.2, D.2.6, D.3.2, D.3.4, F.3.2, F.4.2.
Keywords & Phrases: Algebraic specification, application languages, term rewriting, automatic tool generation.
Note: To appear in M. Wirsing (ed.) *Algebraic Methodology and Software Technology; Proceedings of AMAST'96*, Lecture Notes in Computer Science, Springer-Verlag, 1996. Invited talk for the *AMAST'96 Education Day*.

1 Introduction

Day to day automation in trade and industry makes little or no use of formal methods. Instead, software construction and maintenance in the real world deals with COBOL, PL/I, IMS or Fortran, and at best with SQL, C++, graphical user interfaces, or CORBA-like middleware. Of all software running in the world, the fraction developed using formal methods is practically equal to zero.

In spite of that, many computer science departments training their students to become professional software engineers, have included formal method courses in their curricula. The reason for this is not only that such courses help to understand the foundations of computing; most departments strongly believe that the use of formal methods will greatly enhance the software development process. But we should ask ourselves whether teaching these classes is not simply a waste of time and energy. Is it not so that students, once they have found a software engineering job in industry, will find it almost impossible to spot opportunities for applying their formal methods skills?

If we take these questions seriously, we have to teach students how to apply their knowledge in practice. We must train them to recognize which situations may be ripe for formalization, and when the cost of using formal techniques will not outweigh the benefits. We have to show students how we tried to use formal methods, how our business partners reacted, and which techniques we mainly used. Wherever possible, we have to involve students in our collaboration with industry, and make them participate actively.

In this paper, we address such questions in the context of the algebraic specification formalism ASF+SDF. We report on our experiences with the use of ASF+SDF in Dutch industry, and on the role students have been playing in these.

1.1 Language Prototyping

Languages are ubiquitous in software engineering: they are used for specification and implementation, as interfaces between components, to access databases, to build user interfaces, and so on. Many software systems are centered around one or more *domain-specific* languages, which are tailored to the specific needs of the system in question and which can be used to extend a system easily with new application software.

It is the aim of ASF+SDF to assist during the design and further development of such languages [2, 14, 7]. It consists of a formalism to describe languages and of a Meta-Environment to derive tools from such language descriptions. Ingredients often found in an ASF+SDF language definition include the description of the (1) context-free grammar, (2) context-sensitive requirements, (3) transformations or optimizations that are possible, (4) operational semantics expressing how to execute a program, and (5) translation to the desired target language. The Meta-Environment turns these into a parser, type checker, optimizer, interpreter, and compiler, respectively.

1.2 ASF+SDF

The Formalism The language ASF+SDF grew out of the integration of the Algebraic Specification Formalism ASF and the Syntax Definition Formalism SDF [2]. An ASF+SDF specification consists of a declaration of the functions that can be used to build *terms*, and of a set of equations expressing *equalities* between terms.

If we use ASF+SDF to define a language L, the grammar is described by a series of functions for constructing abstract syntax trees. Transformations, type checking, translations to a target language L', etc., are all described as functions mapping L to, respectively, L, Boolean values, and L'. These functions are specified using conditional equations, which may have negative premises. In addition to that, ASF+SDF supports so-called *default*-equations, which can be used to “cover all remaining cases”, a feature which can result in significantly shorter specifications for real-life situations [7]. Specification in the large is supported by some basic modularization constructs.

Terms can be written in arbitrary user-defined syntax. In fact, an ASF+SDF signature is at the same time a context-free grammar, and defines a fixed mapping between sentences over the grammar and terms over the signature. Thus, an ASF+SDF definition of a set of language constructors specifies the concrete as well as the abstract syntax at the same time. Moreover, concrete syntax can be used in equations when specifying language properties. This smooth integration of concrete syntax with equations is one of the factors that makes ASF+SDF attractive for language definition.

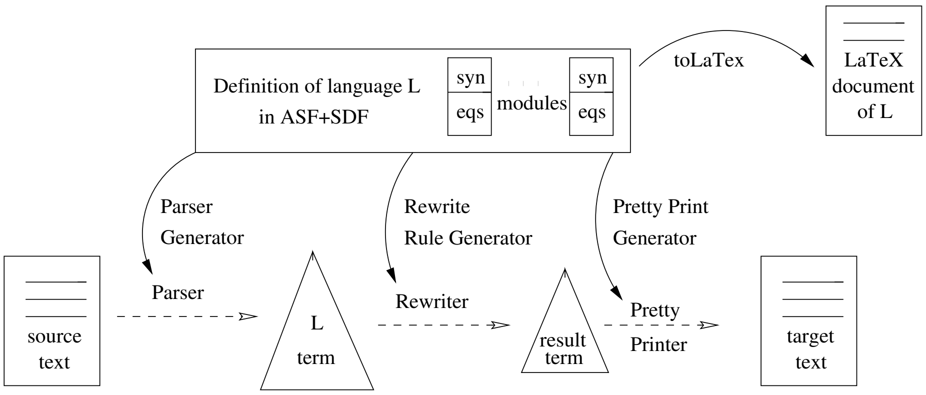

Fig. 1. A language definition for L in the ASF+SDF Meta-Environment.

The Meta-Environment The role of the ASF+SDF Meta-Environment [14] is to support the development of language definitions, and to produce prototype tools from these. It is best explained using Figure 1. A modular definition of language L, generates *parsers*, which can map L-programs to L-terms, *rewriters*, which compute functions over L-programs by reducing terms to their normal form, and *pretty printers*, which map the result to a textual representation. In the Meta-Environment, the generators are invisible, and run automatically when needed. The derived pretty printer can be fine-tuned, allowing one to specify compilers to languages in which layout is semantically relevant (e.g., COBOL) [6].

This pattern gives rise to a series of language processors, with a functionality as specified in the language definition. Basic user-interface primitives can be used to connect the processors to an integrated L-specific environment.

The ToLaTeX facility of the ASF+SDF Meta-Environment encourages the language designer to write his or her definition as a *literate* specification.

2 Applying ASF+SDF in Practice

The typical industrial usage of ASF+SDF is to support the design of a domain-specific language. The formalism is used to write a formal language definition, and the Meta-Environment is used to obtain prototype tools. Once the language design is stable and completed successfully, the prototype tools can be re-implemented in an efficient language like C, although there are also examples in which the generated prototype is satisfactory, and re-implementation is not even considered.

The underlying observation is that language design is both critical and difficult, and that it should not be disturbed by implementation efforts in a language like C. At the same time, prototype tools are required during the design phase to get feedback from language users. ASF+SDF helps to obtain these tools with minimal effort, by executing the language definitions, and by offering a number of generation facilities.

This requires an extra investment during the design phase, since ASF+SDF enforces users to write a thorough language definition. The assumption is that this investment will pay for itself during the implementation phase, an assumption confirmed by the various projects carried out so far, such as the ones discussed below.

2.1 Structure Definition in Compiler Production

An example of the use of ASF+SDF for the development of a successful industrial language is the "full Structure Definition Language" (fSDL) [13]: a language for data structure descriptions, developed in the ESPRIT compiler generation project COMPARE. It has been used by Associated Computer Experts B.V. to develop their commercially available CoSy Compiler Suite. Using the primitives of fSDL, a compiler developer can define complex data types such as lists, tables, and graphs, together with functions for manipulating these. fSDL definitions are compiled to C.

During the design of fSDL, a *flattener* and the compilation to C were specified using ASF+SDF. The language design phase was a ponderous process, with various parties involved, during which a series of changes and language releases were issued. The ASF+SDF code describing the language took approximately 3000 lines, corresponding to roughly one person-year. However, once the design was completed, the final C implementation could be written very easily, and required no significant changes in fSDL.

2.2 Interlocking in Railway Applications

An example of the use of ASF+SDF for prototyping tools comes from the safety guaranteeing systems as used by the Dutch Railways (NS) [11]. To ensure that signals along railway yards are turned to green only when this is safe, devices called *Vital Processor Interlockings* (VPIs) are used. VPIs are programmed using the *Vital Logic Code* language (VLC). In collaboration with Utrecht University and NS, a project was carried out to verify safety properties of VPIs by expressing VLC in process algebra. Several tools were prototyped using ASF+SDF, mostly for transforming VLC programs, or for translating them to propositions that can be given as input to a tautology checker. The ASF+SDF specifications, covering a total of 400 lines of code, were used as the basis for an efficient implementation in C. An interesting observation is that NS had initially estimated that building a complete implementation would cost them one person year. Using ASF+SDF, a full functional prototype was constructed in only seven person-days.

2.3 RISLA: A Specification Language for Financial Products

RISLA is a language for describing *financial products* such as *mortgages*, *interest rate swaps*, etc. It has been developed by Bank MeesPierson and software house Cap

Volmac. In principle, a product defines a scheme of agreements between a bank and its customers. A product may have several parameters, such as the principle amount, the interest rate, whether or not irregular redemptions are allowed, etc. An instance of such a scheme is called a *contract*. Several software systems running in the bank, such as the financial administration or the risk analysis programs, require information about the various financial contracts established. For a large part, this information is characterized by the *cash flows* each product generates. A cash flow is a list of (payment, date) pairs.

Originally, a product was described informally by a financial engineer, and then implemented in COBOL by the software engineer. However, products are subject to changes, and new products are introduced on a regular basis. This involved a severe maintenance effort. Moreover, it was difficult to guarantee the correspondence between the informal description and the implementation. Therefore, in 1992, the software engineers involved with interest rate products started to think about the specification language RISLA; a language that should be readable by financial experts on the one hand, and that should permit automatic COBOL-code generation on the other hand.

A first version of RISLA The starting point for RISLA was the observation that all elementary operations could be defined as functions in a COBOL library. The first step was the semi-formal description of several existing products in terms of these functions.

Using ASF+SDF a formal language definition for RISLA has been extracted from these example descriptions [1]. Furthermore, some ideas about information hiding and structuring have been formulated in an object-oriented manner.

Since then a product has been regarded as an object. It has an internal state, and is parameterized by its *contract data*, which are given a value upon contract creation. *Information methods* can be defined to retrieve data from a contract; *registration methods* can be used to update the state of a contract (for example, to register a redemption). Auxiliary methods can be defined, which are invisible for the outside world, and methods can be parameterized. At present, recursion is not allowed.

RISLA product descriptions can be translated to COBOL. The current compiler is written in C, although the COBOL generator has also been specified in ASF+SDF as part of a M.Sc. project [16].

By now about 40 financial products have been defined in RISLA. Their generated COBOL code is in daily operation. The product descriptions, however, have become longer and longer; low level computations are specified in great detail for each product again. This showed that the specifications were not yet at the right abstraction level. As a consequence, only a small group of RISLA experts could read them, and the communication with the financial experts still occurred on the basis of the informal descriptions.

Modular RISLA In order to solve this problem the *Upgrade RISLA* project was started in the summer of 1995. The project team consisted of people from various disciplines: accountants, software consultants and people who knew about ASF+SDF.

To reduce the size of RISLA specifications, RISLA was extended with modularization features to construct products from separate *components*. Like a product, a component has contract data and methods. Components represent common notions such as interest payment schemes. Products and components can import other components, and, if necessary, rename the contract parameters or the method names.

The import/renaming mechanism has been provided with a semantics by an *expansion* function translating a modular specification into a *flat* one. The specification of the syntax of RISLA, the syntax of the modularization mechanisms, a type checker and the expansion procedure takes about 100 pages of ASF+SDF. The obtained flat form can be compiled further to COBOL using the existing RISLA to COBOL compiler.

Currently, a library of components has been developed, and about 10 products have been specified in terms of these components. A financial expert can use the components as simple building blocks to construct new products.

Risquest: A Language for Questionnaires The RISLA-environment should also support the smooth introduction of new products. Therefore the language Risquest has been developed, which can be used to specify a *questionnaire*, i.e., a list of interactive questions by which information about the new product is extracted in a structured manner from a user. While filling in a questionnaire, a first product specification is built up by *selecting* components from the library corresponding to the answers given. Comments are gathered as free text, in case the questions are not sufficiently detailed. After answering the questions, the resulting product description can be fine-tuned; the comments can be formalized by, for example, adapting renamings of certain components. Currently, approximately 20 Risquest procedures have been implemented at MeesPierson.

To run the questionnaire, Risquest is translated into TCL/TK. The specification of the syntax of Risquest, some type checking, part of the syntax of TCL/TK, and the translation takes about 40 pages of ASF+SDF. The pretty printer generator [6] was used to conform to TCL/TK's strict layout conventions.

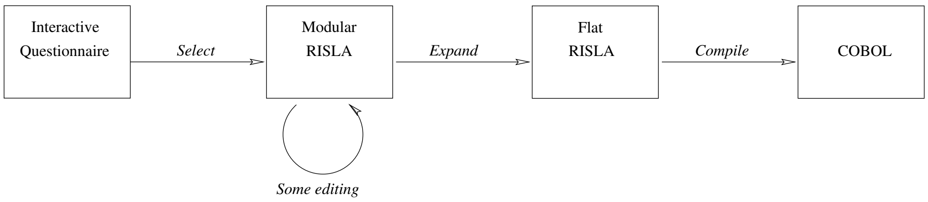

Fig. 2. From Questionnaire via RISLA to COBOL.

Assessment In Figure 2 an overview is given of the environment that combines the questionnaire formalized in Risquest, the modular RISLA obtained from selecting the components according to the answers given, the corresponding flat RISLA, and the resulting COBOL. This environment for financial products meets its requirements; financial engineers can construct their own product specifications, from which COBOL can be generated. Software experts are only needed for the maintenance of the environment (the library of components, the compiler to COBOL, etc.), but are not needed as intermediary between the users and the RISLA specifications.

Only a small part (about one third, or 6 person-months) of the work of the *Upgrade RISLA* team was dedicated to the development of the environment using ASF+SDF. For a great deal these 6 months were spent on engineering matters that come with the use of the ASF+SDF Meta-Environment on a problem of such a large scale, and were caused by the fact that also for the ASF+SDF-experts many aspects were rather new.

The modeling of the financial knowledge itself, such as the definition of the component library and the concrete questionnaire, took most of the effort.

2.4 Renovating Legacy Systems

Unlike the work described in the other sections of this paper, application of ASF+SDF to the renovation of software systems should be labeled as *work in progress* rather than as work already completed. However, we have just (January 1996) embarked on a large project dealing with this subject, in collaboration with several industrial partners including the Dutch ABN-AMRO bank, and we feel that this application is too interesting to leave undiscussed in a paper on industrial applications of ASF+SDF.

The problem at hand involves the analysis, cleaning-up and reconstruction of a large suite (25,000 programs, 30M LOC) of mainframe-based COBOL applications. The two main problems currently studied are conversions between COBOL dialects and identification and correction of software errors related to the "year 2000."

In principle, all techniques available in the ASF+SDF Meta-Environment can be applied to these renovation problems. More specifically, we intend to develop new techniques and tools to support impact analysis, data flow analysis, parsing, pretty printing, code uniformization, goto-elimination, etc.

However, some size problems have to be tackled first. Take, for instance, the COBOL grammar itself. Currently, we have an SDF version of the COBOL grammar of 25 pages. Using this as the basis for any formally defined operation on COBOL programs will lead to unwieldy specifications. Simplification of the COBOL grammar is thus a primary task here. Clearly, applying formal techniques to COBOL is not an activity that has received much attention in the research community, hence we have to start from scratch. On the positive side, we already have a pretty printer for a COBOL subset and we expect that the documentation tools from [6] can be used to build typesetting tools for full COBOL.

Another size problem concerns the amount of information that can be gathered during program analysis. It may, for instance, turn out that program slicing (see [17]) cannot be applied due to the size of the programs to be analysed. We are currently investigating whether more traditional techniques for data flow analysis can be applied, although their results are less precise than those of slicing. A topic here is that we need insight in the data flow of complete applications, as opposed to a single program. This implies that we need to analysed not only individual COBOL programs but also the JCL scripts that combine these COBOL programs into a complete application.

In addition to these size problems, several research questions have to be addressed. Many renovation activities consist of an analyse-regenerate cycle: first information is gathered about a particular source module and then it is regenerated (with the purpose of performing some improvement, e.g., producing another dialect, repairing date-related problems, and the like). In all these activities the question arises how the link between the regenerated code and the original source code can be maintained. For transformations expressed using rewriting, such links can be derived fully automatically using origin tracking [8]. We will investigate which adaptations are necessary to apply origin tracking to the COBOL migration and year 2000 problems.

A last topic directly related to ASF+SDF deals with logging editing actions. Since the state of the art in system renovation is not a fully automated process, the standard mode of operation of a renovation system will be a mixture of fully automatic subtasks and human interventions. In order to enable the complete replay of the whole renovation process, all human interventions have to be captured in *editing scripts* that can be executed automatically later on. Currently, the ASF+SDF Meta-Environment does not support such an editor command language.

3 Concluding Remarks

Education The general pattern displayed in the various projects is the following. Domain-specific languages are important in industrial practice and the use of a formal method such as ASF+SDF helps a language developer to improve the language design significantly, and to obtain prototype tools in a fast and simple manner.

It is easy to convince students as well as potential industrial users of the validity of this claim. At the University of Amsterdam, ASF+SDF[5] has been used in various classes, covering such topics as algebraic specification, software engineering, and compiler construction. The industrial case studies are discussed in such classes, and help the students to recognize similar situations in which formal methods could be applied. Students are encouraged to do their M.Sc. project in the context of ASF+SDF. Many students express interest in carrying out a case study in industry, which has resulted in a series of practical results [1, 15, 10, 20, 5]. We will discuss one of these in more detail.

First Result Consultants (FRC), is a small software house specialized in the field of information systems. Together with two students, FRC initiated a research project investigating the opportunities for automatic code generation from the company's proprietary object-oriented design models (so-called *FRC Models*). These are expressed using a graphical modeling language. The students used ASF+SDF to describe the FRC Models formally and to develop a prototype of a code generator translating FRC Models to *Access* database commands [4]. Both FRC and the students were satisfied with the results and one of the students has been hired to build a production implementation in C++ on the basis of the ASF+SDF specification.

Industrial Participation Cooperation between industry and universities is not without its problems. Managers are under pressure to minimize the risks their companies take, whereas it is a researcher's job to try new and not-yet-proven technology. Moreover, there is a conflict between the company's desire to keep their competitors ignorant of improvements in technology it paid for, and the researcher's ambition to write a paper reporting on this interesting case study. Some of these cultural differences between academia and business partners are covered by [19]. Nonetheless, when these are overcome, industries may gain much more efficient software construction methods, and

[5] An account of the use of ASF+SDF for teaching formal methods is given in [9].

computer scientists will obtain an opportunity to identify the most relevant problems in their field of research – and to what extent their theories help to address these.

Returning to more ASF+SDF-specific issues, the main factor contributing to the acceptance of ASF+SDF is that the formalism is executable, and can be used to actually build something. In some cases, this was the main reason to start using ASF+SDF. Improving the efficiency of the derived prototype tools will increase the industrial applicability, and will lead to further cost reduction if it could eliminate the need to build the final re-implementation in a language like C.

Another point is that in all projects the industries paid academic people to help them during language design or to write prototype tools. It is not (yet) the case that the companies train their own people to start using ASF+SDF. However, there are already several examples of companies hiring people specifically for their ASF+SDF expertise.

Research Questions Language design and tool building based on term rewriting technology has a high potential, and further research will help to increase the acceptance and industrial applicability. We mention two topics that need to be addressed:

- The efficiency of the prototypes is for a large part determined by the efficiency of the rewrite machinery used. Some results from functional programming can be used here, but the problem is both simpler – there is no need for dealing with λ-expressions – and more complicated – the pattern matching can be arbitrarily deep. Current research has led to a prototype ASF2C compiler [12].
  One technical problem is how and when to output the resulting term: for realistic specifications (such as the one for fSDL) the normal form may be several megabytes in size, and in order to keep the prototype's runtime memory small such results should be written to file as early as possible. But when can one be sure that part of a term is in normal form, and will not be changed later? It may be that results of combining rewriting with I/O are of help here [18].
- Another critical issue is the openness of the prototypes built. In an industrial setting, a tool never runs alone. Instead, it will require application-specific data from other components, or functionality available in libraries modeled in other languages. At present, we are investigating to what extent the TOOLBUS[6], a software bus based on process algebra [3], can help to address these problems.

Acknowledgments Thanks to T.B. Dinesh, Jasper Kamperman, Wilco Koorn, Eelco Visser, Machteld Vonk, and Pum Walters. The authors acknowledge Bank MeesPierson and the Centre of Excellence for Risk Management, Finance division Cap Volmac, for the set up of the Upgrade Risla project and their permission to publish the results of the project in Section 2.3. We thank Jan Boef from ABN-AMRO for his encouragement to publish the work described in Section 2.4.

[6] A demonstration of the TOOLBUS will be given during this AMAST conference.

References

1. B. R. T. Arnold, A. van Deursen, and M. Res. An algebraic specification of a language for describing financial products. In M. Wirsing, editor, *ICSE-17 Workshop on Formal Methods Application in Software Engineering*, pages 6–13. IEEE, April 1995.
2. J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. The ACM Press in cooperation with Addison-Wesley, 1989.
3. J. A. Bergstra and P. Klint. The TOOLBUS coordination architecture. In *Proceedings of the First International Conference on Coordination Models, Languages and Applications*, Cesena, Italy, April 15–17 1996. to appear.
4. F. Bonsu and R. Oudejans. Graphic generation language — automatic code generation from design. Master's thesis, Programming Research Group, University of Amsterdam, 1995.
5. M. 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. In M. van den Brand, A. van Deursen, T. B. Dinesh, J. F. Th. Kamperman, and E. Visser, editors, *ASF+SDF '95: A Workshop on Generating Tools from Algebraic Specifications*, Technical Report P9505, pages 29–52. Programming Research Group, University of Amsterdam, May 1995.
6. M. van den Brand and Eelco Visser. Generation of formatters for context-free languages. Technical Report P9506, Programming Research Group, University of Amsterdam, 1995. To appear in ACM Transactions on Software Engineering Methodology.
7. A. van Deursen, J. Heering, and P. Klint (eds.). *Language Prototyping, An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific Publising Co., 1996. To Appear.
8. A. van Deursen, P. Klint, and F. Tip. Origin tracking. *Journal of Symbolic Computation*, 15:523–545, 1993. Special Issue on Automatic Programming.
9. T. B. Dinesh and S. M. Üsküdarlı (Eds.). Teaching formal methods using the ASF+SDF Meta-environment. Technical report, CWI and University of Amsterdam, July 1994. Proceedings of the NSF Workshop on Teaching Formal Methods, URL: ftp:// ftp.cwi.nl/ pub/gipe/drafts/ TFM.ps.Z.
10. J.N. Entken. A prototype of a simulator for hydraulic systems. Master's thesis, Programming Research Group, University of Amsterdam, 1993.
11. J. F. Groote, S. F. M. van Vlijmen, and J. W. C. Koorn. The safety guaranteeing system at station Hoorn-Kersenboogerd. Technical report, Department of Philosophy, Utrecht University, 1995.
12. J. F. Th. Kamperman and H.R. Walters. Lazy rewriting and eager machinery. In J. Hsiang, editor, *Rewriting Techniques and Applications, RTA'95*, volume 914 of *Lecture Notes in Computer Science*, pages 147–162. Springer-Verlag, 1995.
13. J. F. Th. Kamperman, T. B. Dinesh, and H. R. Walters. An extensible language for the generation of parallel data manipulation and control packages. In Peter A. Fritzson, editor, *Proceedings of the Poster Session of Compiler Construction '94*, 1994. Appeared as technical report LiTH-IDA-R-94-11, university of Linköping; Full version as CWI Report CS-R9575.
14. P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.
15. L.H. Oei. Pruning the search tree of interlocking design and application language operational semantics. Technical Report P9418, Programming Research Group, University of Amsterdam, 1994.
16. M. Res. A generated programming environment for RISLA, a specification language defining financial products. Master's thesis, Programming Research Group, University of Amsterdam, 1994.

17. F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.
18. H. R. Walters and J. F. Th. Kamperman. A model for I/O in equational languages with don't care non-determinism. In *Workshop on Abstract Data Types ADT'95*, Lecture Notes in Computer Science. Springer-Verlag, 1996.
19. D. Weber-Wulff. Selling formal methods to industry. In J. C. P. Woodcock and P. G. Larsen, editors, *FME'93: Industrial Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.
20. C. Zaadnoordijk. An ASF+SDF specification of a query optimizer for a RDBMS. Master's thesis, Programming Research Group, University of Amsterdam, 1994.