The Evolution of Implementation Techniques in the ASF+SDF Meta-environment

P. Klint

Department of Software Technology, Centre for Mathematics and Computer Science, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
Programming Research Group, University of Amsterdam, P.O. Box 41882, 1009 DB Amsterdam, The Netherlands

1 Introduction

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[1] 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 [HK94]. An overview of the Meta-environment is given in [Kli93].

2 The past: the GIPE I and GIPE II projects

Developments started in close cooperation with, in particular, INRIA in France, in the ESPRIT projects GIPE I[2] and GIPE II[3] projects. Both projects were successful, and this makes it all the more interesting to start our investigations into the evolution of implementation techniques by having a look at the original project proposal, from which I quote:

[1] GIPE—Generation of Interactive Programming Environments.

[2] The GIPE I project was carried out in the period 1985–1989 with the following partners: BSO (The Netherlands), CWI (The Netherlands, with University of Amsterdam as subcontractor), INRIA (France), and SEMA-METRA (France).

[3] The GIPE II project was carried out in the period 1988–1993 with the following partners: BULL (France), CWI (The Netherlands, with University of Amsterdam as subcontractor), GIPSI (France), INRIA (France), Planet (Greece), PTT Research (The Netherlands), and SEMA-METRA (France). Initially ADV-ORGA (Germany) was involved but early on they left the project and their tasks were taken over by the University of Darmstadt (Germany). The University of Linköping (Sweden) joined the project during the last two years.

Proceedings of ASF+SDF95. A workshop on Generating Tools from Algebraic Specifications. May 11 & 12, 1995, CWI, Amsterdam, M.G.J. van den Brand, A. van Deursen, T.B. Dinesh, J.F.Th. Kamperman & E. Visser (eds.) Technical Report P9504, Programming Research Group, University of Amsterdam

> *The main objective of this project is to investigate the possibilities of automatically generating interactive programming environments from language specifications. An "interactive programming environment" is here understood as a set of integrated tools for the incremental creation, manipulation, transformation and compilation of structured, formalized, objects such as programs in a programming language, specifications in a specification language, or formalized technical documents. Such an interactive environment will be generated from a complete syntactic and semantic characterization of the formal language to be used. In the proposed project, a prototype system will be designed and implemented that can manipulate large formally described objects (these descriptions may even use combinations of different formalisms), incrementally maintain their consistency, and compile these descriptions into executable programs.*

The main steps in the plan were as follows:

- Establish a common software environment.
- Define common interfaces between components.
- Define a set of common examples to be used as "benchmarks" for language specification formalisms.
- Design and implement the environment generator.
- Generate environments for the selected examples.

In retrospect this was a sensible approach that worked out well. Continuing with the "establishment of a common environment" the proposal reads:

> *This task is concerned with the construction, installation and documentation of a standard software environment as a point of departure for experimenting with and making comparisons between language specification techniques. The necessary elements of this–UNIX-based– software environment are: efficient and mutually compatible implementations of Lisp and Prolog, parser generator, general purpose algorithms for syntax-directed editing, general purpose pretty printing algorithms, software packages for window management and graphics etc. Most of these elements are already available or can easily be obtained; integrating these components into one standard software environment will be the major effort.*

With the advantage of hindsight, we now know that the basic assumptions in this description were wrong. The components we needed were not available and all integration efforts turned out to have been systematically underestimated. Even if components were available, like for instance the syntax-directed editor of the Mentor system [DGHKL84], or the window systems that existed at that time, their integration amounted in nearly every case to a complete re-implementation.

Another observation is that the proposal was based on too static a view on technological developments and the response of researchers to them: the assumption was that once a certain technology has been integrated it can stay in place. In reality, technological development goes so fast, that new techniques have to be assessed on a continuous basis and their integration requires an ongoing, significant, effort. The world outside the project creates new technological opportunities, but it is in most cases the pressure from *inside* the project by individual researchers that causes transitions to new techniques. The fear to stay behind or to miss the connection

with a new development that is perceived as important are strong motivations for this behaviour. Let's be honest, which researcher does not recognize the child in him- or herself that wants to play with new and exciting toys?

In the following paragraphs, I will discuss some specific technological developments that affected the project.

2.1 Implementation language

In the beginning, we were all convinced that one flexible, dynamic, implementation language should be used for the implementation of all components of the system. That language should support run-time type checking, garbage collection, run-time generation of programs, and the like. INRIA's Mentor system had been implemented in Pascal and the lack of garbage collection and the limitations of Pascal's type system were felt as serious drawbacks. CWI's work on "monolingual environments" [HK85] was based on an earlier, positive, experience with the Summer programming language [Kli80] that supported dynamic typechecking and automatic garbage collection. Our collective experiences were thus consistent with each other.

INRIA's research group for VLSI design had just completed a small, portable, Lisp implementation (fashionably called "LeLisp") that they intended to use for building a design and test environment for VLSI chips. Since Common Lisp was still in its infancy (and we all feared that the resulting language would be huge), it was only natural to adopt LeLisp as a common implementation language. In retrospect I think this was the wrong choice:

- Given the simplicity and portability of LeLisp we believed that it would become a freely available, widely used, popular, language. However, as more and more features were added to LeLisp it became less simple, and thus less portable. When the further development and distribution of LeLisp were taken over by INRIA's subsidiary ILOG, there also came an end to the free availability of LeLisp, in particular for commercial users.

- Although a module concept was added to LeLisp later on, as a language it remained weak in structuring large programs. At the time that the modules were sufficiently mature (in version 16), the language had become so incompatible with previous versions that the project could not afford the cost of converting all software to the new version.[4] At that time we were, of course, completely trapped in the use of a non-standard language without further support or perspective.

In retrospect, our language-centered approach was wrong. The requirements listed above were, obviously, valid ones, but the conclusion that a single language should provide them was untenable, as exemplified by the adoption of Prolog as a second implementation language early in the project. In Section 2.5 we will see that LeLisp has been used as the central glue to tie components together. This was a requirement that was *not* on our list. From this perspective, LeLisp was certainly a better choice than, for instance, C, but LeLisp was not particularly equipped for component integration either. As a result, a substantial amount of work was needed for the integration of each new component,

Given the technology that was available at that time, it would have been better to leave the choice of an implementation language to the implementors of individual components and to standardize only at the level of data representations. Each component could then be implemented as a separate (UNIX-level) program (using an appropriate implementation language) that exchanges data with other programs according to predefined file formats.

[4] A conscious decision *not* to follow a new development!

2.2 User-interfaces

The impact of the developments sketched in the previous paragraph is quickly forgotten if we look at the breath-taking sequence of systems for constructing user-interfaces that have played a role in the project:

- The *Brown Workstation Environment* (BWE) and in particular its window manager *A Screen Handler* (ASH): This was one of first window systems for UNIX workstations. All ASH library call were interfaced with LeLisp.
- *SunWindows*: One of the first window managers for Sun workstations built by Lucasfilm Inc.
- *X-windows*: the now standard window system pioneered at MIT. The Xt library was interfaced with LeLisp.
- *Graphical Objects*: an object-oriented construction kit implemented in LeLisp.
- *Motif*: a library on top of the basic X-windows library. The Motif library was interfaced with LeLisp.
- *UI manager*: a stand-alone user-interface manager developed at CWI and UvA, based on Motif, implemented in C.

Since each new user-interface system typically included several hundreds of entry points, the amount of integration work per new system was substantial.[5] Early on, an attempt was made to define a set of *abstract functions* to interact with different windowing systems. This attempt was a failure since no consensus could be reached on what should be *excluded* from this interface.

2.3 Rewriting engines

While the rapid developments in the area of user-interfaces came, to a large extent, from outside the project, I now wish to focus on developments coming completely from the inside: the evolution of techniques for implementing rewrite systems. Here, the list of efforts is also impressive:

- A first ASF typechecker was implemented in the Summer language (in fact, this had already been done before the start of the GIPE project).
- In an effort to identify existing technology for term rewriting, we performed experiments with O'Donnell's Equation Interpreter and with C-Prolog [HK86]. The former system looked interesting since it performed extensive preprocessing on the specification in order to generate efficient rewriting code. Several small ASF specifications were compiled (by hand) to the input language of the Equation Interpreter and to Prolog.

  At that time[6], the preprocessing times needed by the Equation Interpreter were preposterous and did not lead to significant improvements in execution time when compared to the straightforward interpretation using C-Prolog. Our obvious conclusion was that Prolog was the preferred implementation vehicle.

[5] The people at INRIA should be credited for this work.

[6] We stopped following the development of the Equation Interpreter after drawing conclusions from our experiments. Later publications show that its implementation has been improved compared to the version we used.

- Several rewrite engines for ASF have been implemented that compile specifications to Prolog. They were different in the precise translation rules being used and in the overall organization of the generated Prolog code [BW89].
  - A scheme described by Drosten en Ehrig [Die89, Hen91].
  - A scheme described by Van Emden/Yukawa, resembling the code we generated in the experiments with the Equation Interpreter [Hen91].
- Prolog implementations are based on unification, while term rewriting only needs matching. Clearly, improvements could be made by further exploiting the specific properties of term rewriting. This has resulted in two approaches that compile specifications to Lisp:
  - A fully compilational approach first described by Kaplan and implemented by Casper Dik in his master's thesis.
  - The Equation Manager that is used today in the ASF+SDF Meta-environment (implemented by Casper Dik).
- In our search for maximal efficiency, we also turned our attention to the compilation of specifications into C. Based on a newly designed abstract machine dedicated to term rewriting (ARM), the first ASF2C compiler translates specifications first to ARM code and then to C [KW93].

Each new implementation resulted in an increase of execution speed with as a result that we currently have—as far as we know—the fastest rewriting techniques in existence.

However, one could, with good reason, pose the question how many different implementations are needed before the "final" one is reached. There are several answers, none of them giving a clue how to avoid such an iterative development process:

- Iterative development is unavoidable since it reflects the learning curve yielding increasingly better insights in the behaviour of term rewriting.
- Initially, we expected to piggy-back on the advances in Prolog compilation technology. However, in order to profit from these advances, the generated Prolog code has to contain more and more annotations to influence the Prolog compiler. These annotations are different for each Prolog compiler, thus slowly blocking the transition to even newer Prolog compilers.
- The interpretation of Horn clauses and term rewriting systems exhibit more and more subtle differences, the closer one looks. The former need full unification while the latter can be implemented with matching only.
- If efficiency is the ultimate aim, one should generate code in the most efficient language around. For portability reasons, it is not attractive to generate assembly language. Therefore, the best compromise seems to be to generate plain C code.

2.4 Lazy/incremental generation techniques

Two forms of incremental behaviour can be distinguished: incrementality of the generator and incrementality of the generated environment. One of the hall marks of the current implementation of the ASF+SDF system is the use of lazy/incremental techniques for the generation of lexical scanners, parsers, and, to a lesser extent, term rewriting systems [HKR90, HKR92] This approach is guided by the following principles:

- *lazy*: only generate those parts of an implementation that are currently needed.
- *incremental*: whenever the input description changes (e.g., the grammar used by the parser generator to generate a parser), remove those parts of the already generated implementation that are inconsistent with the new input description. Rely on lazy expansion of the adapted implementation to further generate the implementation for the new input description.

This approach hides most details of generating implementations and makes the ASF+SDF Meta-environment easy to use even for people who are unaware of implementation aspects. However, as part of the current analysis, the following observations should be made:

- It is not easy to combine lazy/incremental program generation and global optimization techniques: the former depends essentially on partial knowledge while the latter needs global knowledge. A typical example is the use of first/follow sets in a parser generator. By using these sets one can generate better parsers but this requires processing the whole grammar which is contrary to the lazy/incremental approach. As a result, implementations generated with a lazy/incremental generator will in most cases be inferior to implementations generated by a more classical generator that can use global information during the generation process.
- A lazy/incremental generator may detect errors in the given input description in a later stage than a conventional generator.
- In lazy/incremental generators a lot of additional bookkeeping is necessary (i.e., dependencies between the current input description and the corresponding, partially generated, implementation). In large applications, the size of this additional information can become prohibitive.
- Due to the bookkeeping just mentioned, lazy/incremental generators are more complex to build (and maintain) than more conventional generators.
- A lazy/incremental generator needs the original input specification. In a fully compiled, stand-alone, generated environment this is unnecessary overhead.

What can we infer from these observations? The current seemless integration of editing specifications and generating implementations was and remains very attractive: editing specifications and the use of the corresponding (generated) implementation can alternate without any further actions being required from the user. It might very well be possible to develop fast, classical program generators, that implement this behaviour with acceptable performance. Such generators would also more naturally support the generation of stand-alone environments.

Incrementality in *generated* environments has been introduced and implemented experimentally as described in [Meu90]. Nearly the same arguments as given above apply to this form of incrementality: (1) the generator becomes more complex; (2) when applications become larger the bookkeeping needed for incremental execution increases and the benefits of the approach diminish due to the overhead of the larger (virtual) memory that is needed.

2.5 Connecting components

From the previous analysis it will be clear that a major question is how to glue all the different components together. We will briefly describe three approaches.

LeLisp as glue. Interfaces have been developed between LeLisp and C, ASH, C-Prolog, mu-Prolog, X, Motif, and the X resource manager. In all these cases LeLisp's external function interface has played a crucial role in establishing connections with (mostly C based) external software: by writing appropriate interface code and by linking the external software with the LeLisp code, an extended version of the LeLisp system is obtained. The following observations can be made about this approach:

- Writing the interfacing code is a repetitive, error prone task.
- The extension mechanism is *static*: all external software has to be included *at link time* in the extended version of the LeLisp system. This results in ever growing core images, unless one creates versions of LeLisp for each desired combination of external packages.
- The rapidly growing memory requirements of the resulting systems led to the desire to *distribute* the implementation over more than one machine. The extension mechanism just described does not support this.

Software IC's and SophTalk. These observations led the INRIA group to start work on more general mechanisms for connecting software components. They chose as metaphor the "integrated circuit" (IC): a building block that implements a certain functionality and that has a number of *pins* for making connections with other IC's. By using appropriate *wires* (or buses) one can build a system as a network of cooperating IC's [Clé90]. Later on, the software IC paradigm evolved into the SophTalk system [BJ93], essentially a library of LeLisp functions for building networks of software IC's. SophTalk also supports *external* IC's: Unix programs that implement an IC and are connected to LeLisp via sockets. Lisp expressions are used to exchange data with the external IC. In this way, distributed client/server applications can be build. The SophTalk system has been used with success in later versions of the Centaur system.

I very much liked the ideas behind the SophTalk approach, but kept my doubts about it. The major criticism was that these IC networks seem to be *too* distributed, and do not permit an easy understanding of the cooperation between IC's since all control has been distributed as well. This was confirmed by the difficulties encountered when trying to understand (and debug) such networks[Dis94].

Connecting Emacs. Most computer users spend most of their time using some text editor or word processor. This explains why the editing facilities offered by a system are a frequent source of complaints and criticism. The editor in the Meta-environment (GSE [Koo92]) was built as a research prototype and we have never considered including elementary features like search, undo, and the like. Recognizing, however, the importance of good editing capabilities for the overall acceptance of the system we decided to perform an experiment and replace the text-editing functions of GSE by Emacs [KB93]. The overall approach is sketched in Figure 1. In the old situation, the Meta-environment was implemented as a single, monolithic, Unix process. In the envisaged, new, situation three components were created, each executing as a separate Unix process:

- A user-interface manager (UIM) built using Motif. All user-interface aspects of the Meta-environment were handled by sending appropriate commands to UIM.
- Epoch, a version of Emacs integrated with X-windows.
- The remaining parts of the Meta-environment including the structure-oriented editing commands of GSE but excluding text-oriented commands.

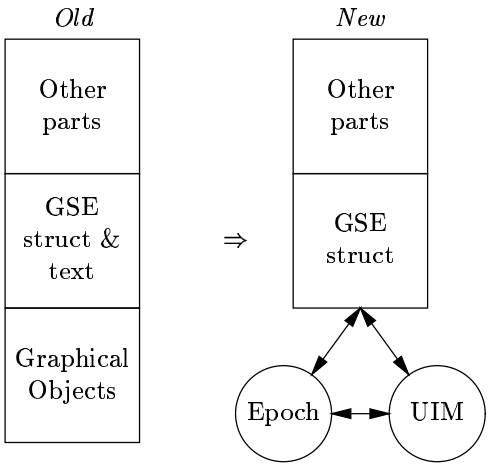

Figure 1: Making the connection with Emacs

With a considerable effort, this new implementation was completed but never worked really satisfactorily for the following reasons:

- The outstanding misjudgement in this operation was that the consequences of moving from a single process implementation to a multi-process implementation were recognized much too late. Initially, design and implementation were focussed on individual components rather than on their cooperation. As a result, synchronization problems started to occur after the design, coding and testing of the individual components were mostly complete. At that stage, these problems had to be solved by ad hoc means that could not guarantee the overall correct behaviour of the cooperating components. A formal specification and simulation of this cooperation [vVVvW94] revealed more, at that moment not yet discovered, communication problems. These observations formed the starting point for the ToolBus approach discussed in Section 3.2.

- In our desire to gradually move from the old implementation to the new one, the protocols between the components were made to mostly simulate the old interfaces between components. However, these old interfaces assumed a shared-memory between components with uniform access costs for all data structures. The new situation was based on a distributed-memory model with non-uniform access costs. As a result, the costs of communication between components were relatively high and the performance of the new, distributed, editor was rather poor.

2.6 Lessons

Goals Initially, we envisaged the generation of environments for "programs in a programming language, specifications in a specification language, or formalized technical documents" (see Section 2). All examples selected as benchmark for the generator were small programming languages focusing on specific features (e.g., static and polymorphic typing, block structure, and goto's.) During the project, the relevance of application languages became clear. Typical examples are special languages for database queries, form-based data-entry, product descriptions, pro-

duction planning, and financial products. As a result, the current Meta-environment is biased towards solving idiosyncracies of the Fortran syntax rather than addressing probably more relevant issues like how to define a syntax that can be used to generate a form-based editor. For these applications it is also important to generate small, stand-alone, environments that can run independently of the Meta-environment.

Technology A long-term research project should try to make itself immune to the rapid development of information technology by assuming a continuously changing rather than a fixed technological infrastructure. In this project, for instance, the bias towards a single implementation language (LeLisp) and towards specific technologies (e.g., user-interface toolkits) was wrong. Since it is very hard to predict which technical trends will become accepted standards, there seems to be no obvious solution for this problem, other than selecting approaches that are as insensitive as possible to technological changes.

System structure The larger the system becomes the more difficult it is to maintain and extend it. Frankly, at the moment nobody dares to touch old code out of fear that modifications will have unexpected effects on other parts of the system. As a result many interesting results have not been incorporated in the system as distributed.

Another development that makes the limitations of the current structure manifest is the desire to move towards a more open, distributed, implementation in order to connect to externally available components (e.g., Emacs)

The overall structure of the implementation and the way components are connected are hence becoming a growing concern.

How many roads? In [Bro74] a "second systems effect" is described. The first time that a team builds a system, many design errors are made out of ignorance. The second time, an effort is made to avoid all previous errors with as outcome an unwieldy, unusable system. The third time, a well-balanced, clean, system can be designed and build. This seems optimistic compared to the accounts given earlier (see Section 2) and this raises the anxious question how many roads we must pass by until ....[7]

3 The present: towards a next generation

The global requirements for the next generation Meta-environment can be distilled from the lessons described in the previous section and are as follows:

- G1 No bias towards a single implementation language.
- G2 Transparent, language-independent, exchange of data between components.
- G3 Explicit, clear mechanisms for describing the control interactions between components.
- G4 Transparent connection with externally available software packages.
- G5 Implementation can be distributed over more than one process and/or computer.

Two aspects of these requirements are now further discussed: data integration (Section 3.1), and control integration (Section 3.2).

[7] Free after Donovan.

3.1 Data integration: GEL and ASFIX

The requirement to have transparent, language-independent, exchange of data between components can be further specialized as follows:

D1 The only data exchanged between components are terms.

D2 Any sharing in a term should be preserved when it is communicated between components.

D3 The representation of terms should be concise, to permit representation and exchange of terms containing millions of nodes.

D4 Terms should be self-descriptive, i.e., they should contain all necessary signature information to permit type-checking.

D5 It should be possible to attach annotations to terms in order represent arbitrary, descriptive, information.

D6 Specifications should also be represented as terms and should be self-descriptive.

Data integration mechanisms satisfying these requirements are provided by the data representation languages GEL and ASFIX. The former is intended for the concise, linear, encoding of arbitrary graphs. The latter will be used for the logical, self-descriptive, representation of terms and specifications.

3.1.1 GEL—Graph Exchange Language

GEL (Graph Exchange Language) [Kam94] provides a means for concisely encoding arbitrary graphs with as important special case: terms with sharing. Basically, GEL provides a dictionary mechanism for introducing abbreviations for arbitrary functions names, and a graph building engine that is programmed via postfix commands. A concise, linear, encoding of graphs is achieved that requires circa one byte per node for large graphs. The encoding and decoding can be done efficiently, making GEL a good representation mechanism for communicating graphs between components in a distributed application.

If we consider a standard definition of the Booleans, then the GEL encoding of the term `true & false` will become:[8]

```
!a0:0=Booleans: "true" -> BOOL
a0
!a1:0=Booleans: "false" -> BOOL
a1
!a2:10=Booleans: BOOL "&" BOOL -> BOOL
a2
```

The three lines marked with `!` introduce abbreviations for the constants `true` and `false` and the `&` function. Each abbreviation has the form

`!`*abbreviation*`:`*arity*`=`*string*

with *abbreviation* the shorthand being introduced, *arity* the symbol's arity (represented as binary number), and *string* an uninterpreted string that may contain arbitrary (even binary) data of arbitrary size. One could, for instance, use an arbitrary bitmap as function name. The commands `a0` and `a1` push the constants true

[8] We use here a textual presentation of GEL, in the implementation a more concise byte encoding is used.

and false on the graph stack. The command `a2` consumes these two constants and replaces them by the desired term.

Two aspects of GEL are *not* illustrated by this simple example. First, an abbreviation has to be introduced only once, but can be reused many times thus contributing to a concise representation. Second, the instructions of the graph engine permit the construction of arbitrary (cyclic) graphs.

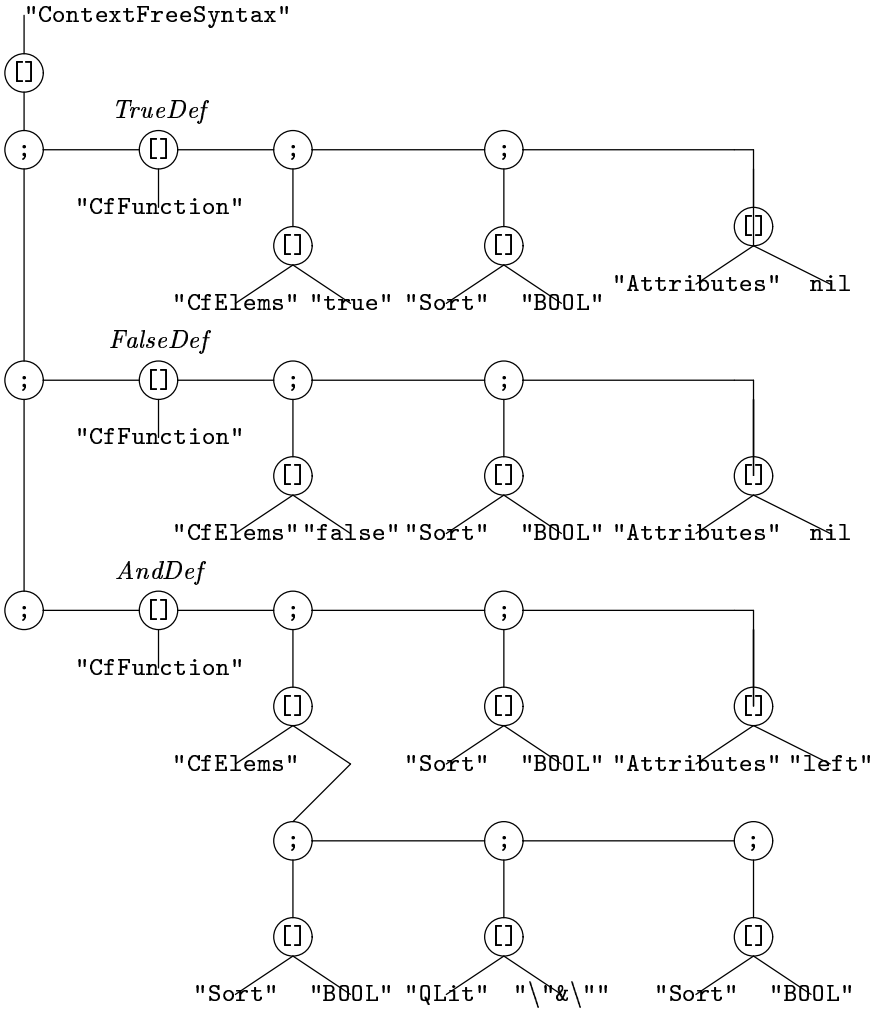

Figure 2: Graph representation of the context-free syntax of the Booleans. The nodes labeled *TrueDef*, *FalseDef*, and *AndDef* are referenced from terms (see Figure 3).

3.1.2 ASFIX—ASF+SDF prefix representation

The syntactic freedom provided by ASF+SDF makes it impossible to use a single, fixed, syntax for each specification. This fact hinders the development of "meta-level specifications": ASF+SDF specifications manipulating ASF+SDF specifications. Such specifications are essential for obtaining concise, readable, descriptions of tools like typecheckers and compilers for ASF+SDF. ASFIX (derived from

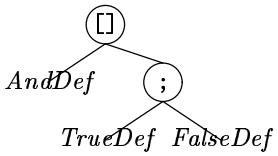

Figure 3: Graph representation of the term `true & false`

ASF+SDF *preFix representation*) is a fixed exchange format for ASF+SDF specifications [Kli94]. The overall representation of specifications is made very simple by representing a *complete* ASF+SDF specification as a single term. This representation is sufficiently self-contained to specify, for instance, the complete type checking of ASF+SDF specifications. A generic annotation mechanism turns ASFIX into a versatile representation and storage medium for tools operating on ASF+SDF specifications.

ASFIX is based on the notion of "ATerms": applicative, annotated term structures. ATerms are either literal strings, or they are constructed using function application ("`[`" and "`]`"), list construction ("`;`") or annotation ("`/`"). An informal definition of `ATerms` is:

- *Literals*: a literal is a term, e.g., `"f"` or `"37"`.
- *Lists*: the empty list `nil` as well as expressions of the form T_1 `;` T_2 are terms. Lists are used to represent sequences of terms. Example: `a; b; c`.
- *Function application*: Only unary function application is defined and has the form `[`T_1 T_2`]`, where T_1 and T_2 are arbitrary terms. For instance, `"f"` is a legal term but `["f" "a";"b";"c"]` as well as `[["g" "1"; ["h" "2"]] "a";"b";"c"]` are also legal terms. The equivalents of these three terms in a more conventional notation would be: `f`, `f(a,b,c)`, and `g(1,h(2))(a, b, c)`. At the position where ordinarily a function name appears, we allow thus an arbitrary term. Nonetheless, we will frequently call such a term the "function symbol".
- *Annotations*: a term T_1 can be annotated with another term T_2. This is written as T_1 `/` T_2 and the result is also a term.

ATerms can be defined as follows.

Module ATerms

imports Literals[3.1.2]
exports
  sorts ATerm
  context-free syntax
    Literal → ATerm
    nil → ATerm
    ATerm ";" ATerm → ATerm {**right**}
    "[" ATerm ATerm "]" → ATerm
    ATerm "/" ATerm → ATerm {**left**}
    "(" ATerm ")" → ATerm {**bracket**}
  priorities
    ATerm "/" ATerm → ATerm > ATerm ";" ATerm → ATerm

The imported module `Literals` contains straightforward definitions and is not shown.

An ASFIX specification is now simply an ATerm of a certain prescribed form. For instance, the SDF section

```
context-free-syntax
  true          -> BOOL
  false         -> BOOL
  BOOL "&" BOOL -> BOOL {left}
```

will be translated into

```
["ContextFreeSyntax"
  ["CfFunction" ["CfEelems" "true"]; ["Sort" "BOOL"]; ["Attributes" nil]];
  ["CfFunction" ["CfElems" "false"]; ["Sort" "BOOL"]; ["Attributes" nil]];
  ["CfFunction" ["CfElems" ["Sort" "BOOL"]; ["QLit" "\"&\""]; ["Sort" "BOOL"]];
                ["Sort" "BOOL"]; ["Attributes" "left"]]]
```

Each context-free function declaration, is represented by `"CfFunction"` followed by three arguments: a list of context-free elements, a result sort, and a list of attributes. Observe, that literals that need to be quoted are represented by `"QLit"`, other literals stand for themselves. The symbol `"Sort"` represents sort names in the original specification. The graph presentation of this fragment is shown in Figure 2.

The ASFIX representation of terms satisfies requirement D4 given earlier: terms should be self-descriptive, i.e., they should contain all necessary signature information to permit type-checking. This is achieved by using *context-free function definitions as function symbols*. Our example term `true & false` will then be represented as follows:

```
[["CfFunction" ["CfElems" ["Sort" "BOOL"]; ["QLit" "\"&\""];
               ["Sort" "BOOL"]]; ["Sort" "BOOL"]; ["Attributes" "left"]]
 [["CfFunction" ["CfElems" "true"]; ["Sort" "BOOL"]; ["Attributes" nil]] nil];
 [["CfFunction" ["CfElems" "false"]; ["Sort" "BOOL"]; ["Attributes" nil]] nil]]
```

Although its textual representation is verbose, the corresponding graph structure is as simple as it should be as is shown in Figure 3. Observe that all terms share the subgraphs *TrueDef*, *FalseDef*, and *AndDef* defined in Figure 2.

3.1.3 Discussion

ASFIX and GEL complement each other. The ASFIX version of a term represents its logical structure and is self-descriptive. It contains many shared subterms. GEL can be used to obtain a concise physical representation of the ASFIX version of the term, taking into account all sharing.

3.2 Control integration: the TOOLBUS

A solution for the control interconnection problem should satisfy the following requirements:

- C1 It has a formal basis and can be formally analysed.
- C2 It is simple, i.e., it only contains information directly related to the objective of control integration.
- C3 It exploits a number of predefined communication primitives, tailored towards our specific needs. These primitives are such, that the common cases of deadlock can be avoided by adhering to certain styles of writing specifications.

C4 The manipulation of *data* should be completely transparent.

C5 There should be no bias towards any implementation language for the tools to be connected. We are at least interested in the use of C, Lisp, Tcl, and ASF+SDF for constructing tools.

C6 It can be mapped onto an efficient implementation.

In [BK94, BK95] a proposal has been made for an architecture called the "ToolBus". We will now briefly summarize the ToolBus architecture (Section 3.2.1) and then we show how it can be used to build a syntax-directed editing environment (Section 3.2.2).

3.2.1 ToolBus—a component interconnection architecture

ToolBus:

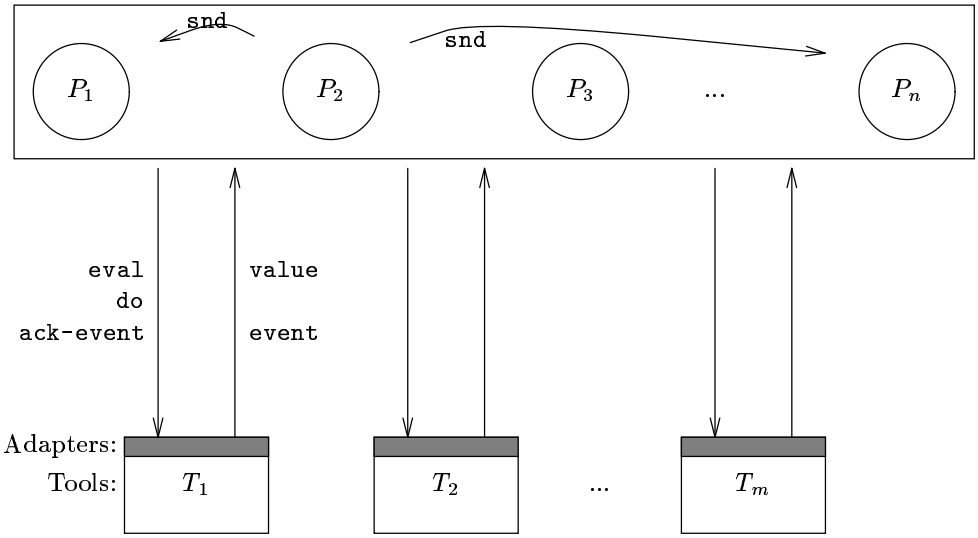

Figure 4: Global organization of the ToolBus

The global architecture of the ToolBus is shown in figure 4. The ToolBus serves the purpose of defining the cooperation of a variable number of *tools* T_i $(i = 1, ..., m)$ that are to be combined into a complete system. The internal behaviour or implementation of each tool is irrelevant: they may be implemented in different programming languages, be generated from specifications, etc. Tools may, or may not, maintain their own internal state. Here we concentrate on the external behaviour of each tool. In general an *adapter* will be needed for each tool to adapt it to the common data representation and message protocols imposed by the ToolBus.

The ToolBus itself consists of a variable number of processes[9] P_i $(i = 1, ..., n)$. The parallel composition of the processes P_i represents the intended behaviour of the whole system. Tools are external, computational activities, most likely corresponding with operating system level processes. They come into existence either

[9]By "processes" we mean here computational activities *inside* the ToolBus as opposed to, for instance, processes at the operating system level. When confusion might arize, we will call the former ToolBus processes" and the latter "operating system level processes". Typically, the whole ToolBus will be implemented as a single operating system level process. This is also the case for each tool connected to the ToolBus.

by an execution command issued by the ToolBus or their execution is initiated externally, in which case an explicit connect command has to be performed by the ToolBus. Although a one-to-one correspondence between tools and processes seems simple and desirable, we do not enforce this and permit tools that are being controlled by more than one process as well as clusters of tools being controlled by a single process.

The ToolBus is programmed by means of **T** scripts providing features like creation, sequential and parallel composition, choice, and iteration of processes, and operations related to tools like execution/termination, connection/disconnection, and sending/receiving evaluation requests.

```
process TOP is
  let Uid : ui, Sid : syn-edit, Eid : int, Filename : str
  in
    execute(ui, Uid?) .
    execute(syn-edit, Sid?) .
    ( rec-event(Uid, edit(Filename?)) .
      create(ED(Uid, Sid, Filename), Eid?) .
      snd-ack-event(Uid, edit(Filename))
    + rec-event(Uid, close(Filename?)) .
      snd-msg(editor(Filename), close) . snd-ack-event(Uid, close(Filename))
    ) * rec-event(Uid, quit) . snd-note(quit) . snd-ack-event(Uid, quit) .
    shutdown("End of editing")
  endlet

#include "ed-defs.tb"

process ED (Uid : ui, Sid : syn-edit, Filename : str) is
    subscribe(quit) .
    ED-STARTUP(Uid, Sid, Filename) .
    ED-COMMAND(Uid, Sid, Filename) * ED-SHUTDOWN(Uid, Sid, Filename)

tool syn-edit is { command = "syn-edit" }
tool ui       is { command = "wish-adapter -script ui-edit.tcl" }
toolbus(TOP)
```

Figure 5: **T** script for editing environment.

3.2.2 A syntax-directed editor

Now we show how an environment for syntax directed editing can be described in which a user can open and close an arbitrary number of syntax-directed editors on different files. The **T** script to achieve this is shown in Figures 5 and 6. This example is an oversimplification but clearly shows an approach that can be used for defining similar, more realistic, systems.

Imagine a toplevel user interface containing three push buttons: `Edit`, `Close`, and `Quit`:

- Pushing `Edit` results in a dialogue asking for a file name. Once the file name has been provided by the user, a syntax-directed editor for the file is created.
- Pushing `Close`, also results in a dialogue asking for a file name in order to identify the editor instance to be closed.

```
process ED-STARTUP (Uid : ui, Sid : syn-edit, Filename : str) is
  let Msg : str
  in
    snd-eval(Sid, edit(Filename)) .
    ( rec-value(Sid, error(Msg?)) .
      snd-do(Uid, displayError(Msg)) . delta
    + rec-value(Sid, ok)
    ) .
    snd-eval(Uid, mk-text-editor(Filename)) .
    ( rec-value(Uid, error(Msg?)) .
      snd-do(Uid, displayError(Msg)) . delta
    + rec-value(Uid, ok)
    )
  endlet

process ED-COMMAND (Uid : ui, Sid : syn-edit, Filename : str) is
  let X : int, Y : int, Bgn : str, End : str
  in
    ( rec-event(Uid, Filename, tree-up) .
      snd-eval(Sid, tree-up(Filename))
    + rec-event(Uid, Filename, tree-down) .
      snd-eval(Sid, tree-down(Filename))
    + rec-event(Uid, Filename, tree-next) .
      snd-eval(Sid, tree-next(Filename))
    + rec-event(Uid, Filename, mouse(X?,Y?)) .
      snd-eval(Sid, mouse(Filename, X,Y))
    ) .
    rec-value(Sid, focus(Bgn?, End?)) .
    snd-do(Uid, setFocus(Filename, Bgn, End)) .
    snd-ack-event(Uid, Filename)
  endlet

process ED-SHUTDOWN (Uid : ui, Sid : syn-edit, Filename : str) is
    ( rec-msg(editor(Filename), close) + rec-note(quit)
    ) .
    snd-do(Sid, close-editor(Filename)) .
    snd-do(Uid, close-editor(Filename))
```

Figure 6: `ed-defs.tb`: auxiliary definitions for editing environment.

- Push `Quit` closes all editors and terminates the execution of the whole environment.

The next question is how to model syntax-directed editing and how to coordinate text editing and syntax-directed editing. We assume the existence of two tools: a user-interface (`ui`) providing text editors and a syntax-directed editing tool (`syn-edit`) providing structure information. The idea is now that a text editor generates events whenever information is needed from the syntax-directed editor. For instance, an "`up`" event from the text editor will lead to a request to the syntax-directed editor to calculate a new focus, representing begin and end point of the text area representing the parent of the current focus. This focus information can then be used to adjust the focus area in the text editor.

In Figure 5 the main parts of the resulting **T** script are shown: a single process `TOP` handles the user-interface described above and creates a new instance of process `ED` for each new editor. It first executes the user-interface tool (`ui`) and the syntax-directed editing tool (`syn-edit`). Next, it handles the cases `Edit` and `Close` in a loop. On receiving `Quit`, this loop is terminated and the whole editing environment is shutdown. The quit operation is implemented by broadcasting a `quit` note to all editor processes.

The process `ED` first subscribes to "`quit`" notes, performs initialization, executes editing commands in a loop, and then ends the execution of this editor instance. The process `ED` is defined by three auxiliary parameterized process definitions `ED-STARTUP`, `ED-COMMAND`, and `ED-SHUTDOWN` shown in Figure 6. Typically, the user-interface tool generates an event, e.g., `mouse(10,35)`, which is then transferred to the syntax-directed editing tool in order to calculate a new focus. The latter is then communicated back to the user-interface in order to highlight the new focus in the text.

A characteristic of this approach is the complete decoupling between user-interface and semantic processing. In a more conventional approach, a so-called *call back* routine will be associated with each user-interface element. When, for instance, a button is pushed by the user, its associated call back routine will be activated to implement the button's behaviour. In the editing example given above we see that:

- Each event is represented abstractly by a term, e.g., `tree-up`.
- The binding between an event and its handler is dynamic rather than static. This binding is determined by the **T** script rather than by the user-interface.
- The user-interface and the associated handlers can (a) run on different machines; (b) be implemented in different languages.

3.3 Discussion

Given the ingredients described in the previous section, the approach taken for re-engineeringing the ASF+SDF Meta-environment can be made more precise. First, a global architecture description can be made describing the required components and their cooperations. Typical ingredients currently being developed are:

- Parser generator.
- Pretty printer generator [vdB93].
- Documentation tool generator [vdBV94].
- ASF+SDF compiler.

- Generic syntax-directed editor.
- Top level user-interface.

The components related to the user-interface are directly implemented in Tcl/Tk [Ous94]. All other components are specified in ASF+SDF. This will yield a running prototype for the next generation Meta-environment. The functional aspects of this prototype can then be assessed before a possible re-implementation of time critical components is considered.

4 The future: towards a ToolFactory

The previous sketch of current work directly related to the implementation of a next generation ASF+SDF Meta-environment can be extended with work directly or indirectly *using* the current system in diverse application areas, for instance,

- Visual editing [Ü94].
- Program transformations.
- Use of and connection with verification and proof systems.
- Simulation of traffic systems.
- Evolutionary development of existing and new formalisms.
- Building Meta-environments for other specification formalisms [vD94].
- Using origin tracking [vDKT93] and program slicing [Tip95] for reverse engineering.

This list of subjects is so heterogeneous that it is unrealistic to expect that the old ambition to generate environments for "programs in a programming language, specifications in a specification language, or formalized technical documents" will ever lead to a single environment generator that can accommodate all the requirements generated by these diverse application areas.

However, on closer inspection they have much in common. They can all be seen as systems that *read*, *write*, *store*, *transform*, *annotate* and *display* terms. For instance, a system for program transformations will read an initial program design in textual form and parse it. Next, transformation rules will be applied to it interactively yielding a new term and a set of proof obligations. The sequence of transformation steps can be seen as a single term that is extended and displayed as controlled by the user.

Another example is an interactive system for reverse engineering. First existing "legacy code" is parsed yielding a term. Next, various tools are executed interactively wich annotate the term with structure information that has been discovered like, e.g., dependence graphs, program slices, and statement classifications. On demand, parts of these annotations will be displayed and used to restructure or translate the original program code.

The common functionalities that can be distinguished are:

- *Parsing and prettyprinting*: to convert programs from textual form to a term and vice versa.
- *Annotation of terms* in order to attach extra information to terms like documentation links, proof obligations, recovered structure information, and the like. Viewing mechanisms are needed to visualize this information.

- *Persistent term storage.* Currently, only the textual version of programs is saved between editing sessions and the corresponding term representation is reconstructed (by parsing) at the beginning of a new session. When the term representation becomes richer than the textual representation, this is no longer possible and the term representation should be stored persistently.
- *Syntax-directed editing* for creating, changing, annotating, and storing terms. In addition, arbitrary tools can be executed on program fragments in the editor.
- *User-interfaces, visualization.* As the range of applications grows, it is no longer possible to provide a single, pre-canned, user-interface. Instead, connections with application-specific user-interfaces and visualization techniques are necessary.
- *Manipulating specifications as data.* There are already several applications that *generate* ASF+SDF specifications (e.g., ASD tools, pretty printer generator). It seems therefore a good idea to develop standard techniques for manipulating specifications as data and to provide facilities for loading generated specifications.

This list of functions sounds, of course, very familiar, albeit somewhat more elaborate than the functionality currently provided. So what is new?

The shift in perspective I propose is the *packaging* of this functionality. Currently, most of the above functions already exist somewhere in the implementation. In most cases, various combinations of them are provided as a single primitive but they cannot easily be identified (or used) as separate functions. A typical example of this phenomenon is the difficulty to re-use matching/rewriting functions (which exist, of course, in the rewrite engine) in a syntax-directed editor for performing either structural searches or systematic structural replacements. This makes it hard to build new combinations of functions that are needed in new applications. A more open system architecture is clearly what is needed. The ToolBus-based approach currently pursued to build the next generation Meta-environment (Section 3.2) is already a first step into this direction.

The longer term perspective is that this line of research will lead to an infrastructure for tool development (a "ToolFactory") in which term formats and the communication between common generators (parser generator, compiler, pretty printer generator, generic syntax-directed editor) and generated tools (parsers, printers, rewrite engines) are shared. By adding new generators and tools, and by combining old and new tools in new ways, a level of flexibility may be reached that might meet the new demands.

Acknowledgements

Part of the work described here is still in progress and has not yet, or only partly, been described in other publications. This is the case for parser generation (Eelco Visser), redesign of the Meta-environment (Pieter Olivier), the ASF2C compiler (Jasper Kamperman, Pum Walters), and visual editing (Dinesh, Susan Üsküdarli). Jan Heering and Eelco Visser commented on drafts of this paper.

References

[BJ93] J. Bertot and I. Jacobs. Sophtalk tutorials. Technical Report 149, INRIA, 1993.

[BK94] J.A. Bergstra and P. Klint. The ToolBus—a component interconnection architecture. Technical Report P9408, Programming Research Group, University of Amsterdam, 1994.

[BK95] J.A. Bergstra and P. Klint. The Discrete Time ToolBus. Technical Report P9502, Programming Research Group, University of Amsterdam, 1995.

[Bro74] F.P. Brooks. *The Mythical Man-Month.* Addison-Wesley, 1974.

[BW89] L.G. Bouma and H.R. Walters. Implementing algebraic specifications. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 199–282. The ACM Press in co-operation with Addison-Wesley, 1989. Chapter 5.

[Clé90] D. Clément. A distributed architecture for programming environments. In *Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environments*, pages 11–21, 1990. Sofware Engineering Notes, Volume 15.

[DGHKL84] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang. Programming environments based on structured editors: the Mentor experience. In D.R. Barstow, H. E. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*, pages 128–140. McGraw-Hill, 1984.

[Die89] N.W.P. van Diepen. SMALL dynamic semantics of a language with GOTOs. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 133–161. The ACM Press in co-operation with Addison-Wesley, 1989. Chapter 3.

[Dis94] S. Dissoubray. Using Esterel for control integration. In *GIPE II: ESPRIT project 2177, Sixth review report.* january 1994.

[Hen91] P.R.H. Hendriks. *Implementation of Modular Algebraic Specifications.* PhD thesis, University of Amsterdam, 1991.

[HK85] J. Heering and P. Klint. Towards monolingual programming environments. *ACM Transactions on Programming Languages and Systems*, 7(2):183–213, 1985.

[HK86] J. Heering and P. Klint. The efficiency of the Equation Interpreter compared with the UNH Prolog interpreter (extended abstract). *SIGPLAN Notices*, 21(2):18–21, 1986.

[HK94] J. Heering and P. Klint. Prehistory of the Asf+Sdf system (1980–1984). In *From Universal Morphisms to Megabytes: a Baayen Space Odyssey*, pages 341–345. Centrum voor Wiskunde en Informatica, Amsterdam, 1994.

[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.

[HKR90] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *IEEE Transactions on Software Engineering*, 16(12):1344–1351, 1990. Also in: *SIGPLAN Notices*, 24(7):179-191, 1989.

[HKR92] J. Heering, P. Klint, and J. Rekers. Incremental generation of lexical scanners. *ACM Transactions on Programming Languages and Systems*, 14(4):490–520, 1992.

[Kam94] J. F. Th. Kamperman. GEL, a graph exchange language. Technical Report CS-R9440, Centrum voor Wiskunde en Informatica, Amsterdam, 1994.

[KB93] J.W.C. Koorn and H.C.N. Bakker. Building an editor from existing components: an exercise in software re-use. Technical Report P9312, Programming Research Group, Univeristy of Amsterdam, 1993.

[Kli80] P. Klint. An overview of the SUMMER programming language. In *Conference Record of the 7th Annual ACM Symposium on Priciples of Programming Languages*, pages 47–55, 1980.

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

[Kli94] P. Klint. Writing meta-level specifications in ASF+SDF. unpublished note, 1994.

[Koo92] J.W.C. Koorn. GSE: A generic text and structure editor. In J.L.G. Dietz, editor, *Conference Proceedings of Computing Science in the Netherlands, CSN'92*, pages 168–177. SION, 1992. Appeared as Report P9202, University of Amsterdam. Available by *ftp* from ftp.cwi.nl:/pub/gipe as Koo92b.ps.Z.

[KW93] J. F. Th. Kamperman and H.R. Walters. ARM, abstract rewriting machine. Technical Report CS-R93-30, CWI, 1993.

[Meu90] E.A. van der Meulen. Deriving incremental implementations from algebraic specifications. Report CS-R9072, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1990. Extended abstract in *AMAST'91: Proceedings of the Second International Conference on Algebraic Methodology and Software Technology*, Workshops in Computing, Springer-Verlag London (1992), pp 277–286. Available by *ftp* from ftp.cwi.nl:/pub/gipe as Meu90.ps.Z.

[Ous94] J. K. Ousterhout. *Tcl and the Tk toolkit.* Addison-Wesley, 1994.

[Tip95] F. Tip. *Generation of Program Analysis Tools.* PhD thesis, University of Amsterdam, 1995.

[Ü94] S. Üsküdarli. Generating visual editos for formally specified languages. Report P9416, Programming Research Group, University of Amsterdam, 1994.

[vD94] A. van Deursen. *Executable Language Definitions.* PhD thesis, University of Amsterdam, 1994.

[vdB93] M.G.J. van den Brand. Generation of language independent pretty printers. Technical Report P9327, Programming Research Group, University of Amsterdam, 1993.

[vdBV94] M.G.J. van den Brand and E. Visser. From Box to TeX: an algebraic approach to the construction of documentation tools. Technical Report P9420, Programming Research Group, University of Amsterdam, 1994.

[vDKT93] A. van Deursen, P. Klint, and F. Tip. Origin tracking. *Journal of Symbolic Computation*, 15:523–545, 1993.

[vVVvW94] S.F.M. van Vlijmen, P.N. Vriend, and A. van Waveren. Control and data transfer in the distributed editor of the ASF+SDF meta-environment. Technical Report P9415, Programming Research Group, University of Amsterdam, 1994.