Tribute to a Great Meta-Technologist
—from Centaur to The Meta-Environment—

Paul Klint
Software Engineering Department
of
Centrum voor Wiskunde en Informatica (CWI),
and
Informatics Institute, University of Amsterdam
`www.cwi.nl/~paulk`

March 6, 2008

Abstract

Gilles Kahn was a great colleague and good friend who has left us much too early. In this paper I will sketch our joint research projects, the many discussions we had, some personal recollections, and the influence these have had on the current state-of-the-art in meta-level language technology.

1 Getting acquainted

Bâtiment 8. *On a sunny day in the beginning of July 1983 I parked my beige Citroen Dyane on the parking lot in front of Bâtiment 8, INRIA Rocquencourt. At the time, the buildings made the impression that the US military who had constructed the premises in Rocquencourt were also the last that had ever used the paint brush. Inside, lived an energetic research family and I was hosted by project CROAP headed by Gilles Kahn. My roommates Veronique Donzeau-Gouge and Bertrand Mélèse helped me find a bureau in a corner in the cramped building and helped to set up a Multics account on the Honeywell-Bull mainframe.*

After some flirtations with computer graphics, software portability and the Unix operating system, I turned to the study of string processing languages on which I wrote a PhD in 1982 [55]. The main topic was the Summer programming language [52] that featured objects, success/failure driven control flow, string matching and composition, and a "try" mechanism that allowed the execution of an arbitrary sequence of statements and would undo all side effects in case this execution resulted in failure.

As part of this work, I got attracted to the question of how the semantics of such languages could be defined [53]. The approach I used was a meta-circular language definition that covered both syntax and semantics. However, this definition was written after the actual implementation had already been completed. Would it not be great if a language definition could be used to *generate* an efficient language implementation?

As described in more detail in [44], Jan Heering and I started the design of a dedicated programming environment for the Summer programming language. This led us to the notion of a *monolingual programming environment* [43] in which the various modes of the environment such as programming, command line execution and debugging were all done in the same language. We were aware of the formidable implementation effort of such a system for a specific language and, in addition to this, Summer had never been designed with that purpose in mind. As already described, we had some experience with language definitions and this naturally led to the idea of a programming environment based on language definitions.

This is how Jan and I became aware of the INRIA work on syntax-directed editing [36, 60], the formal definition of ADA [37], the language definition formalism Metal [50], and the abstract syntax tree manipulation system Mentor [39, 38, 40]. This work was motivated by Gilles's earlier work on semantic aspects of programming languages [47, 48, 51]. The best way to study this work was to pay a visit to the CROAP (Conception et Réalisation d'Outils d'Aide à la Programmation) team at INRIA, which was headed by Gilles. This is precisely what I did in July 1983. A lucky coincidence was that Tim Teitelbaum and his two PhD students Thomas Reps and Suzanne Horwitz were spending their sabbatical in Rocquencourt. This gave me the opportunity to compare three systems: the Mentor system (Gilles and coworkers), an early version of the synthesizer generator (Tim Teitelbaum and Thomas Reps), and Ceyx (a Mentor-like system built by Jean-Marie Hullot on top of Jerome Chailloux' LeLisp system). This comparison appeared as [54].

2 The GIPE projects

> **Take the money and run!** *Phone call from Gilles early 1984: "Paul, did you hear about this new ESPRIT program? Shouldn't we submit a proposal and take the money and run?"*

2.1 GIPE Proposal

And indeed, by the end of 1984 we submitted a proposal for the project *Generation of Interactive Programming Environments* or GIPE[1] for short. The prime contractor was SEMA METRA (France) and the partners were BSO (a Dutch software house that is currently part of ATOS ORIGIN), Centrum voor Wiskunde en Informatica (Netherlands) and INRIA (France). The objectives and envisaged approach were neatly summarized in the proposal:

> *The main objective of this project is to investigate the possibilities of automatically generating interactive programming environments from a language specification. 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 following steps are required to achieve this goal:*
>
> - *Construction of a shared software environment as a point of departure for experimenting with and making comparisons between language specific techniques. The necessary elements of this – Unix-based – software environment are: efficient and mutually compatible implementations of Lisp and Prolog, a parser generator, general purpose algorithms for syntax-directed editing and prettyprinting, software packages for window management and graphics, etc. Most of these elements are already available or can be obtained; the main initial effort will be to integrate these components into one reliable, shared software environment.*

[1] We never settled on a proper pronunciation of this acronym.

- *A series of experiments that amount to developing sample specifications-based on different language specification formalisms, but initially based on inference rules and universal algebra–for a set of selected examples in the domain of programming languages, software engineering and man-machine interaction. The proposed formalisms have well-understood mathematical properties and can accommodate incremental and even reversible computing.*
- *Construction of a set of tools of the shared environment to carry out the above experiments. It will be necessary to create, manipulate and check (parts of) language specifications and to compile them into executable programs. The tools draw heavily upon techniques used in object-oriented programming (for manipulation of abstract syntax trees), automatic theorem proving (for inferring properties from given specifications to check their consistency and select potential compilation methods), expert systems (to organize the increasing number of facts that become known about a given specification) and Advanced Information Processing in general (man-machine interfaces, general inference techniques, maintenance and propagation of constraints, etc.)*
- *The above experiments will indicate which of the chosen formalisms is most appropriate for characterizing various aspects of programming languages and interactive programming environments. These insights will be used in constructing a prototype system for deriving programming environments from language specifications. The envisioned "programming environment generator" consists of an integrated set of tools and an adequate man-machine interface for the incremental creation, consistency checking, manipulation and compilation of language specifications.*

By performing some hype-aware substitutions (syntax $\mapsto$ model, generation $\mapsto$ model-driven, elements $\mapsto$ components) this vision is still relevant today. As in each proposal we had to oversell our ideas and we indeed needed GIPE (1985-1989), and its sequel project GIPE II (1989-1993) to create a proof-of-concept of this vision. In the GIPE II project, the companies GIPSI (France), Bull (France), Planet (Greece), PTT Research (Netherlands), and the research partners TH Darmstadt (Germany), University of Amsterdam (Netherlands) and PELAB (Linkoping Sweden) joined the team.

2.2 The importance of GUIs

> **Disoriented mice.** *Demonstration sessions are a must for any subsidized research program and ESPRIT was no exception. As part of the yearly ESPRIT conference we gathered—with many colleagues from all over Europe who participated in other ESPRIT projects—in an underground parking lot of the Berlaymont building in Brussels. The parking garage had been turned into an exposition center but the smell of cars was still clearly present. Our two Sun workstations booted-up well but at the stage that the window system was up and running and interaction with the mouse was needed, everything messed up. Incompatible mouse drivers? A hardware error? After two hours of hectic discussions and experiments we discovered the cause of the problem. At that time of early optical mice, the specific grid on the mouse pad was used to determine the mouse's coordinates and simply switching the two mouse pads solved the problem. Or was this a case of overexposure to exhaust fumes after all?*

Gilles had from early on recognized the importance of a proper user-interface. His preoccupation with the user-interface was based on an earlier disappointing experience when the Mentor system was being demonstrated to a high-ranking official from a US government agency. The nifty thing to be demonstrated was that the knowledge of the abstract syntax of a program could be used to skip complete subtrees during a search operation. However, it turned out to be impossible to get this nice idea across because the official kept asking "where's the user-interface?".

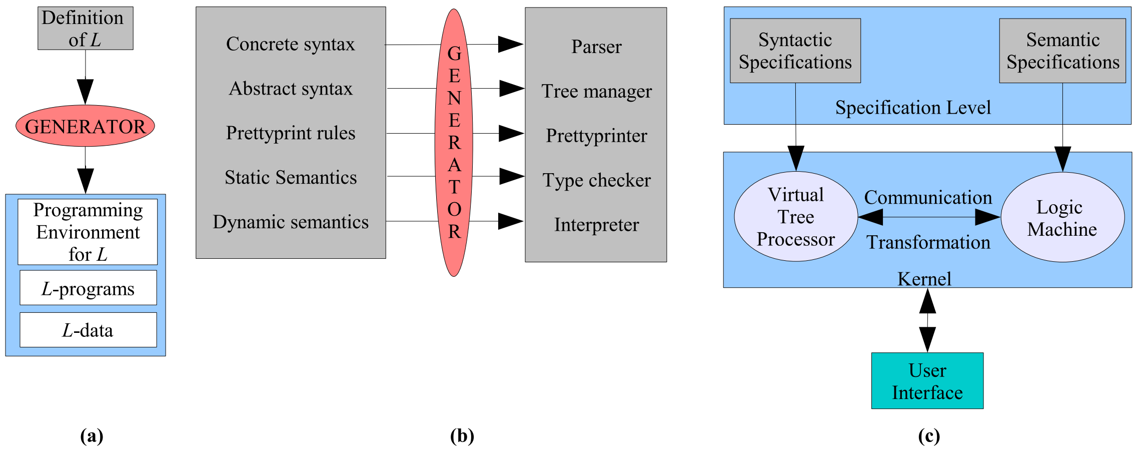

Figure 1: Early architectural designs of the GIPE system: **(a)** End-user view of an L-environment for an arbitrary language L; **(b)** Relation between language definition and generated environment; **(c)** Global architecture.

In fact, during the GIPE project we had the same experience while demonstrating an early prototype of our syntax-directed editor to the board of directors of BSO, our Dutch commercial partner at the time. From our perspective everything was present in the demo: a parser generator, a parser, a syntax-tree manager, and a prettyprinter. All these tools were based on a well-understood theory and implemented with a lot of hard work. However, we learned the hard way that the most important part was still missing: a colorful user-interface that could attract the attention of the board.

It will come as no surprise, that user-interfaces have played an important role during and after the GIPE projects.

2.3 GIPE results

The initial architecture of the environment generator was described in [28]. Figure 1 gives some snapshots of that design. Clearly visible are the generator-based approach and the internal representation of programs as trees.

It is amazing how much effort is often required to achieve goals described in innocently looking sentences in a project proposal, like “Most of these elements are already available or can be obtained; the main initial effort will be to integrate these components into one reliable, shared software environment”. Recall, that the project started in the pre-X-windows era and we have experimented with a lot of now long forgotten systems: Brown Workstation Environment, LucasFilm, LeLisp Virtual Window System, and Graphical Objects.

Another fundamental choice that we made early on in the project was to use LeLisp [23] as the implementation language. LeLisp was a nice and flexible language ideal for prototyping and code generation. However, this decision turned out to be a major problem towards the end of GIPE II. LeLisp was transferred to the INRIA spinoff ILOG and we were stuck with a LeLisp version that was no longer maintained.

There have always been two different approaches in the GIPE projects: a “Dutch” approach and a “French” approach. As a Francophile, I liked Gilles’ laissez-faire style that today would be called “just-in-time”. Several of the younger members in the Dutch team preferred to have things organized weeks in advance. Not surprisingly this led to occasional excitement whether things would be ready for the review, the annual report, the demonstration or whatever. I can testify here, that things were always ready—just-in-time. The other differences in approach will be discussed below.

Initial results of the project were reported in [42] and several other ESPRIT-related conferences. The main outcome of the GIPE projects was the *Centaur system* that was the promised proof-of-concept environment generator [29, 14]. It consisted of:

- The Virtual Tree Processor (VTP): a database for storing abstract syntax trees [61].
- Specification formalisms: Metal (syntax), PPML (prettyprinting) [65], TYPOL (static and dynamic semantics) [33, 26, 49], SDF (syntax) [41], ASF (static and dynamic semantics) [6], and ASF+SDF (the integrated version of the latter two) [6, 34].
- Various editors.
- A user-interface.

In addition to the design and implementation of the system itself various case studies such as for Mini-ML[27, 24], Pico [6], LOTOS [58], Eiffel [1], Esterel [9], Sisal [2], POOL [6] and others were carried out.

After so many years, I should confess that the system was a true Centaur: half human and half horse. On top of the common infrastructure (LeLisp, VTP, Graphical Objects) actually two subsystems were built: a French system using Metal, PPML and TYPOL versus a Dutch system consisting of ASF+SDF and the Generic Syntax-directed Editor (GSE). SDF was also used in the French subsystem for some experiments.

I always found that the French subsystem was ahead of us in terms of inspiring examples and user-interface integration, while we were struggling with issues like incremental parser generation, efficient execution of rewrite rules and understanding how modularization should work. In hindsight, the major results of the project as a whole were:

- The use of natural semantics to specify language semantics.
- The use of user-defined syntax and rewriting techniques to specify language semantics.
- A wide range of implementation techniques that showed the feasibility of these approaches.

At the end of GIPE II the interests of the partners started to diverge further and further. Gilles' team was moving in the direction of the interactive editing of proofs [69, 12] and converting proofs to text [31]. See [22] for an annotated bibliography. On the Dutch side, we were more interested in pursuing the original vision of a programming environment generator; this is documented in [42, 56, 34].

3 An Ongoing Scientific Debate

> **A soldering job.** *In the same parking-lot-turned-into-exposition-centre as the year before we encountered communication problems. How to connect our two workstations in order to exchange relevant software needed for the demonstrations? Gilles and I ended up lying on the floor soldering a null-modem in order to make the connection. It did work, but we were without any doubt the most unqualified electricians in the exposition centre.*

One of the major benefits of the cooperation between the partners in the GIPE projects were the discussions on topics of common interest and the different views and arguments that were exchanged. Gilles was a passionate researcher and had outspoken opinions on all matters relevant to the project. I will briefly describe them below.[2] This scientific debate was the corner stone of the success of the projects. *European cooperative research would be better off if this kind of liberal, scientific debate would be more cherished by the European policy makers.*

[2] Disclaimer: this paper focuses on work resulting from the GIPE projects and mostly ignores other related work.

3.1 Monolingual versus domain-specific languages

As mentioned earlier in Section 1, Jan Heering and I were coming from a "monolingual" background and we wanted to unify as much as possible. Looking at the Mentor approach, we observed that different languages were used for partially overlapping purposes:

- Metal defined concrete syntax, abstract syntax and a mapping from the former to the latter.
- PPML defined a mapping from abstract syntax to a language of boxes.
- Mentol, Mentor's command language, contained constructs for matching and control flow that were also present in Metal and PPML.

This approach has the advantage that each step is explicit and that there is more opportunity for manual tweaking in order to handle complex situations. The disadvantage is, however, that the specification writer has to be aware of several representations and mappings between them. We opted for a more integrated approach in SDF [41] and made the following assumptions:

- There exists a "natural" context-free grammar for the language we want to define. In this way the grammar does not contain idiosyncrasies caused by the use of specific parsing technologies.
- There is a fixed mapping between concrete and abstract syntax.
- Grammars are modules that can be composed.
- A default prettyprinter can be derived from the context-free grammar but can be overridden by user-defined preferences.

Of course, this approach places a tremendous load on the implementation but it leads—in my opinion—to a higher-level specification. So in the case of SDF, the monolingual approach worked well. However, the evolutionary pressures are such that opportunities, desires and needs to introduce specialized formalisms are abundant. Today, we have—in addition to ASF+SDF—dedicated formalisms for component interconnection, relational calculus, prettyprinting, and module configuration. From time to time I wonder if some of them can be eliminated .... Gilles' view that specialized formalisms are unavoidable was certainly the more realistic one.

3.2 Strings versus Trees

With a background in string processing languages, it is not surprising that we had a certain bias in favor of a textual (re)presentation of programs, while Gilles always took the abstract syntax tree as central representation. This resulted in the following differences in point of view:

- In the parsing model, we opted to stay as close to the parse tree as possible, see Section 3.2.1.
- At the specification level, we wanted to have user-defined syntax and give the user complete textual freedom in the concrete textual notation of functions and datatypes. See Section 3.2.2.
- In the editing model, we took the user's text as the primary source of information. This is opposed to the view that the actual text entered is parsed, converted to an internal tree structure, and then presented to the user as the result of prettyprinting that tree. This is explained in more detail in Section 3.2.3.

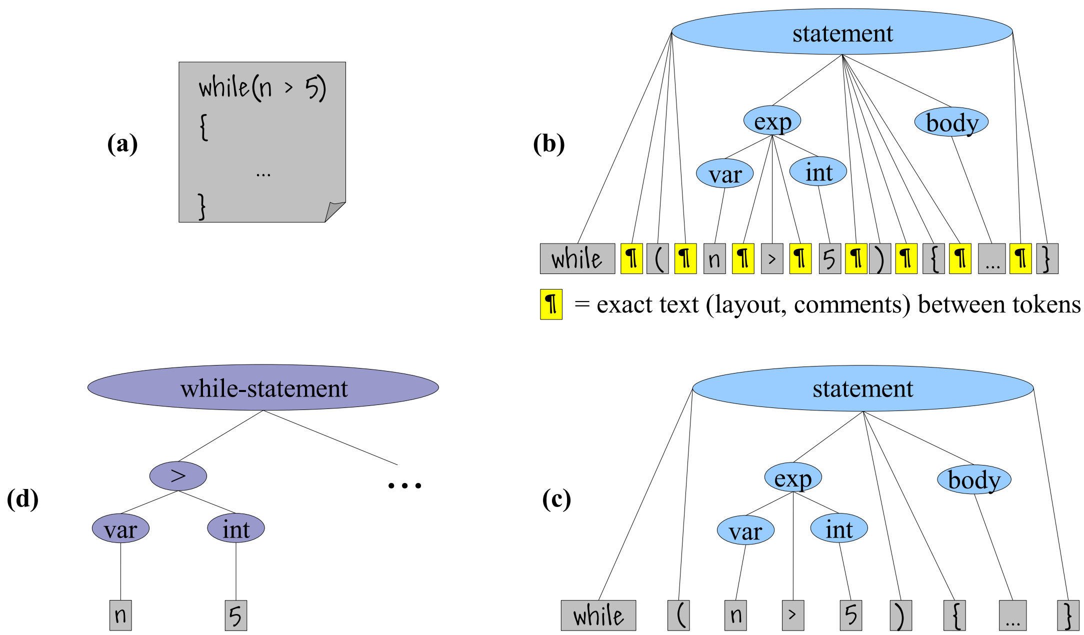

Figure 2: Structured representation of source text: **(a)** Source text; **(b)** Full parse tree; **(c)** Conventional parse tree; **(d)** Abstract syntax tree.

3.2.1 Parse Trees versus Abstract Syntax Trees

What is the right abstraction for representing and manipulating programs? Syntax trees are the structures that are the result of syntax analysis (also known as parsing) of some program text. However, syntax trees exist in various flavors.

Parse trees are a faithful description of the manner in which a text has been been derived from its grammar. A *full* parse tree contains all the textual details of the text including whitespace, comments and the like. See Figure 2(b). Since there is no information loss, it is possible to literally reconstruct the original text from the parse tree. A *conventional* parse tree is a full parse tree with all the layout symbols removed, see Figure 2(c). In this case, the source text can be mostly reconstructed but all layout symbols have to be guessed by a prettyprinter.

Abstract syntax trees omit the textual representation but focus on structure. As shown in Figure 2(d), the abstract syntax tree for a while-statement is a tree node labeled “while-statement” with two children: an expression and the body of the while-statement. From such a *shallow* abstract syntax tree, the source text can only be recreated by means of prettyprinting (see Section 3.2.3). However the original layout (indentation, spacing, comments) is lost. The global advantage of abstract syntax trees is that they require less memory space.

The debate about the proper program representation was always centered around the question how “deep” the abstract syntax tree should be, in other words, how big could the distance between a parse tree and the corresponding “deep” abstract syntax tree be. Suppose that a language contains various iteration constructs, like a for-loop, a while-do-loop, do-until loop, etc. A “deep” abstract syntax tree could map all loop variants on a single, general, loop node. This has the advantage that a single semantic rule can handle all the different cases at once. The disadvantage is that it is no longer possible to know the construct in the original program or to reconstruct the original source text in any way.

Centaur mostly used shallow abstract syntax but had the functionality to build deep syntax trees when desired. In the developments after GIPE, we have focused our attention more and more on software renovation projects where it is mandatory to reconstruct the original source code even after source code transformations. So we moved closer to the source text and now use full parse trees as structured representation of programs.

3.2.2 User-defined syntax

Not surprisingly, our string bias led us to believe that the user should be free to write specifications in a domain-specific notation. In a way, this was our answer to the phenomenon of domain-specific languages. Why write `and(true, or(false, true))` while `true and (false or true)` looks more natural? Writing a compilation rule for an if-statement as

```
compile(if $Test then $Series1 else $Series2 endif) = ...
```

looks more appealing than using a strict prefix notation to represent program fragments. Note that `$Test`, `$Series1` and `$Series2` are meta-variables that represent program fragments. A real example—that has already transformed millions of lines of COBOL code—is the following [20]:

```
addEndIf(IF $Expr $OptThen $Stats) =
IF $Expr $OptThen $Stats END-IF
```

In COBOL, the `THEN` and `END-IF` keywords are optional in if-statements. The above rule inserts `END-IF` keywords in if-statements in order to increase readability and maintainability. Using abstract syntax, a dozen prefix functions would be needed to describe this rule while the version that uses concrete syntax is (almost) readable by the average COBOL programmer. This is another argument in favour of full parse trees.

We decided to aim for an approach where every function or datatype in a specification is in fact a mini-language which defines its own concrete syntax. Of course, these mini-languages should be composable into larger languages and so on. This is a very flexible approach that treats a simple datatype like the Booleans or a complete programming language like Java or COBOL in a uniform manner. The consequences of this decision, however, were staggering:

- Since language grammars had to be composed, we needed a parsing approach that permits grammar composition. Since none of the standard approaches supported this,[3] we started a journey, that has still not ended, in the area of Generalized LR parsing [59, 70, 13, 45].

- Since specification rules contain applications of user-defined grammar rules, it is impossible to define one, fixed, grammar for our specification formalism. As a consequence, we need a two-stage approach: first collect all user-defined grammar rules, generate a parser for them, and then parse the rules in the specification.

In addition to the conceptual challenges, it was also a major problem to implement this in a fashion that scales to really large cases. At several points in time I have thought that the decision to provide user-defined syntax in this way was fatal to our part of the project. Gilles was intrigued by this approach, started a brief study to add it to TYPOL but after a short while concluded that it was not practical and moved ahead to new interesting problems. Since he was not interested in syntax at all, this was probably a wise decision and avoided a lot of complications.

For us, the balance is different. Today we are one of the unique systems that provide fully general user-defined syntax and the major benefit is that program transformation rules can be written in a way that is as close as possible to ordinary source text. This helps in the acceptance of program transformations. However, we have still not solved all problems related to this approach. For instance, since we are working in the domain of full context-free grammars, it remains a challenge how to handle ambiguous grammars.

[3] General context-free grammars are compositional in the sense that they can be combined and form again a context-free grammar. Such a composition can be affected by interferences due to name clashes. Popular subclasses like LL(k) or LR(k) are not compositional at all: the combined grammar may require extensive restructuring in order to satisfy the requirements of the specific subclass.

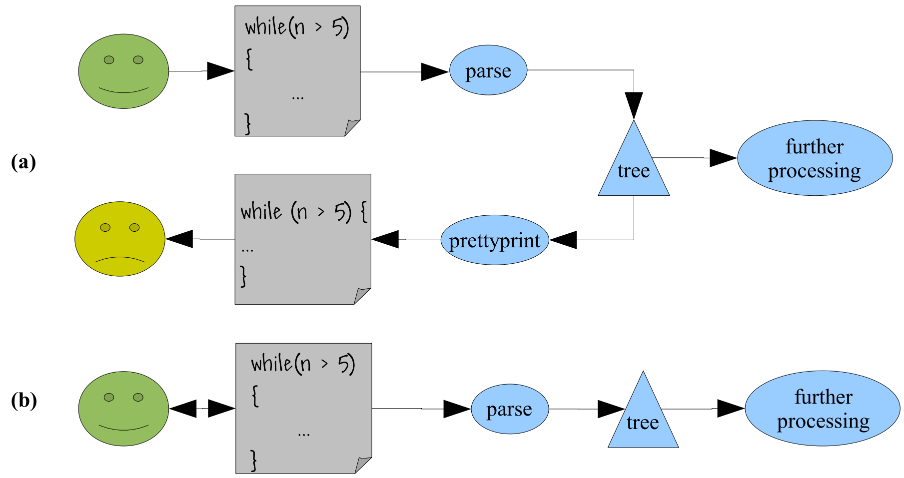

Figure 3: Two editing models: **(a)** User types in source text and sees prettyprinted text; **(b)** User only sees source text as typed in.

3.2.3 From PPML to Pandora

> **Don't touch that link!** *In the early days of the World Wide Web we were all excited about the new information infrastructure that was emerging. Gilles showed us the latest version of the Mosaic browser. "Don't touch that link since it connects all the way to Australia!" he said with concern in his voice when I tried it. At that time of expensive dial-in connections, today's cheap broadband connectivity was impossible to foresee. Later, Gilles played a key role in the transfer of the W3C consortium from CERN to INRIA.*

As mentioned above, during the editing of programs we took the text as entered by the user as the primary source of information. After a textual modification, the text was reparsed and the internal tree structure was updated. The textual image as presented to the user remained exactly as the user had typed it in. This is shown in Figure 3(b). In the standard editing model used by Centaur, the text was parsed, converted to a tree structure, and then presented to the user as the result of prettyprinting that tree. This is shown in Figure 3(a).

A *prettyprinter* takes an abstract syntax tree and transforms it into text according to user-defined rules; the subject had been pioneered by Oppen [67]. At INRIA there was already extensive experience with the subject in the form of the Prettyprinting Meta-Language PPML [65]. PPML used a notion of "boxes" that originates from [32] and provides operators for the horizontal and vertical composition of text blocks. PPML provides matching constructs to identify language constructs in the abstract syntax tree, case distinctions, and construction recipes to build box expressions for specific language construction.

Due to our text-oriented view on editing we had no need for a prettyprinter and have lived without one for many years.

PPML was a typical case where our monolingual view clashed with Gilles' view on the use of specialized languages. The PPML constructs mentioned (matching, case distinction, term construction) were also present in the formalisms that performed semantic processing on the syntax tree (be it TYPOL or ASF+SDF) and this duplication of concepts was not appealing for us. Many years later, we gave up ignoring prettyprinting and built several prettyprinters based on the Box language [21, 19]. In our most recent prettyprinter, Pandora, all matching, case distinction and construction of boxes is done in ... ASF+SDF. So here is at least one exceptional case, where monolingualism prevailed.

3.3 From Virtual Tree Processor to ATerms

The Virtual Tree Processor (VTP) [61] was a database system for abstract syntax trees. Given a signature describing the names of the constructor functions, as well as their number and type of arguments, the VTP allowed the construction of arbitrary, type-correct, trees over the defined signature.

In addition to functions for the creation, access and modification of trees, the VTP also provided functionality for creating, maintaining and merging *cursors* (or "paths" in VTP terminology) in each tree. Regarding internal data representation and functionality, the VTP was not unlike a present-day XML processor. The main difference was that the VTP only provided an API (Application Programmer's Interface) for manipulating trees. As a result, programs manipulating trees were restricted to LeLisp although later a C++ version was completed. A serialized form of the trees that could easily be shared by programs written in other languages was missing.

In today's Meta-Environment there is a similar need for manipulating and storing trees. We have taken a textual (and also a binary) representation for trees as starting point and they can be used and exchanged by programs written in arbitrary languages. The resulting format (ATerms and ATerm library [17, 18]) is, in many ways, simpler than the original VTP functionality. One distinctive feature is that it provides (and maintains) maximal subterm sharing, thus considerably reducing the size of large syntax trees.

3.4 Component architecture

Over the years Centaur had evolved into quite a large code base of 200.000–300.000 lines of (mostly LeLisp) code. There were three forces that made us aware of the fact that we needed to reflect on a more component-based approach:

- The French team was more and more interested in connecting external parsers (for instance, for parsing proof traces) and tools (for instance, provers and proof checkers).
- The Dutch team became more and more concerned about the modularity and maintainability of the code base.
- We were all aware of the need to write components in other programming languages than LeLisp; this was partly driven by the availability and support problems with LeLisp that we all were expecting (see Section 2.3).

And, as usual in this cooperation, both teams identified the same problem but ended-up with completely different solutions. Dominique Clément proposed a notion of "software IC", a software component that could be connected with other components via communication channels, not unlike hardware ICs. This approach evolved into Sophtalk [25, 30, 46], a basic, messaging-based, infrastructure for distributed programming. Quoting the Sophtalk website [68]:

> *Sophtalk is a set of tools that enable one to program the interaction between objects following an event model of communication. Sophtalk is an autonomous LeLisp system that provides facilities for programming communication between objects and processes. The system is composed of three packages:* stnode, *a multicast communication mechanism;* stio *an extension of the standard LeLisp asynchronous and synchronous i/o mechanisms; and* stservice, *a mechanism offering interprocess communication at the shell and LeLisp levels.*

The Dutch team had, at the same time, been experimenting with the partitioning of the system in independently executing parts. The primary objective was to increase modularization and to start experiments with writing components in different languages. The initial project to build a new editor from existing components was a disaster [57]. All parts were implemented and tested individually and worked well in isolation. But, when put together, deadlock after deadlock manifested itself. This was a strong incentive to take concurrency seriously and has resulted in the ToolBus coordination architecture [7, 8] that is still in use today. The basic idea of the ToolBus is to have a central, programmable, "software bus" that is used to connect components that may be written in different programming languages, but adhere to a fixed protocol and exchange data in a fixed format (i.e., ATerms, see Section 3.3).

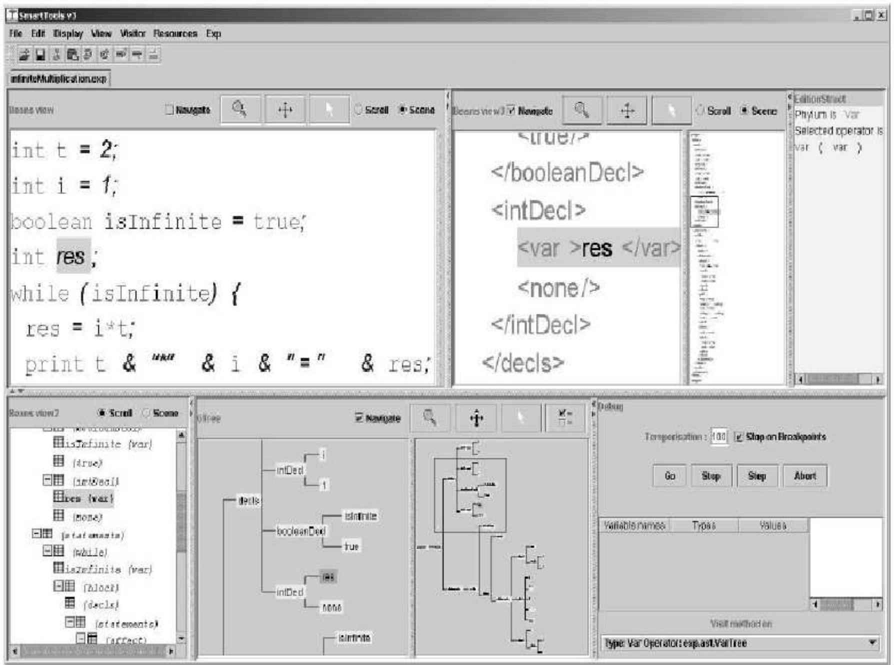

Figure 4: User interface of SmartTools

3.5 Other topics

In addition to the topics already discussed above, the cooperation in the GIPE projects has also been a catalyst for research in a wide range of other areas that were of common interest:

- Origin tracking [11, 35].
- Incremental evaluation [64, 3].
- Generic debugging [10, 66].

I refer the interested reader to the references for further discussions on each of these topics.

4 The post-GIPE Era

4.1 The French Side

The many roads to Route des Lucioles. *Living in a country where winters can be long and dark, I liked the occasional meetings in Sophia-Antipolis where Gilles was working since the mid-eighties. On one of the occasions that he picked me up from a hotel in Antibes he confessed that he was participating in a local competition to find as many new routes towards the INRIA premises as possible. I never recognized the route we took or how we managed to reach Route des Lucioles, on every successive visit.*

4.1.1 CROAP and OASIS

The CROAP project at INRIA was stopped in 1998 and research on generic programming environments was continued in the Oasis project under direction of Isabelle Attali who had, along with others, earlier

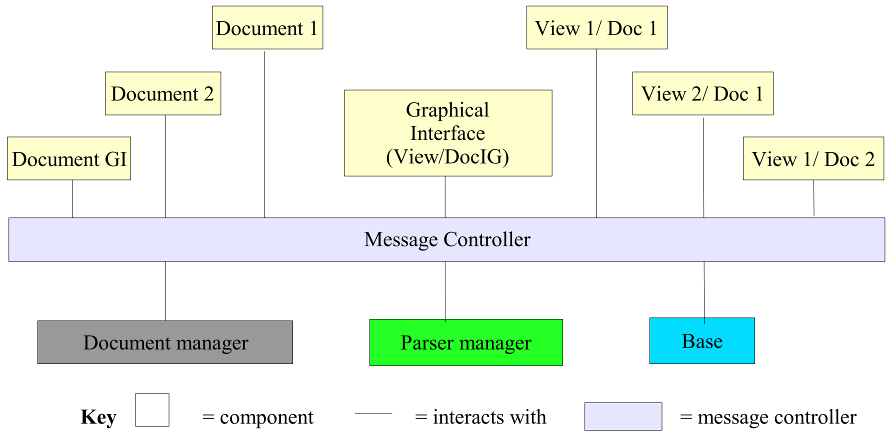

Figure 5: Architecture of SmartTools

worked on the incremental evaluation of TYPOL specifications. In this project, Didier Parigot developed the SmartTools system [5, 4] that can be seen as a second generation Centaur system with a strong emphasis on the use of XML as its tree representation mechanism. The user-interface is shown in Figure 4. It can provide several, simultaneous, views on the same document. The architecture is shown in Figure 5. Note that the user-interface itself is defined in a separate document (Document GI) and that the communication between components is achieved via an asynchronous message infrastructure.

Today, the SmartTools system focuses on domain-specific languages and software factories. Since the interests of the Oasis project gradually moved towards security analysis and smart cards, the SmartTools system has never become a primary focus of the project.

Tragically, Isabelle and her two children died in the 2004 tsunami, while on holiday with her family in Sri Lanka.

4.2 Other impact of the Centaur legacy at INRIA

Many ideas from Centaur still survive in subsequent research activities at Inria. I will briefly mention some clear examples.

4.2.1 AXIS

The AXIS[4] is concerned with verification of information systems and web sites. They are applying Natural Semantics to help specify, check and maintain static semantics of Web sites and more generally of Web documents. Former GIPE team members Thierry Despeyroux (of TYPOL fame) and Anne-Marie Vercoustre are working in this project.

4.2.2 MARELLE

The MARELLE[5] is developing an environment for the computer-supported verification of mathematical proofs. The overall goal is to ensure the correctness of software. Examples are a graphical user-interface for the Coq prover (as already experimented with in GIPE) and the certified implementation of various algorithms. MARELLE is headed by Yves Bertot who was also on the GIPE team.

[4] User-Centered Design, Analysis and Improvement of Information Systems, see `http://www-sop.inria.fr/axis/`.
[5] Computer aided verification of proofs and software, see `http://www-sop.inria.fr/marelle/index_eng.html`.

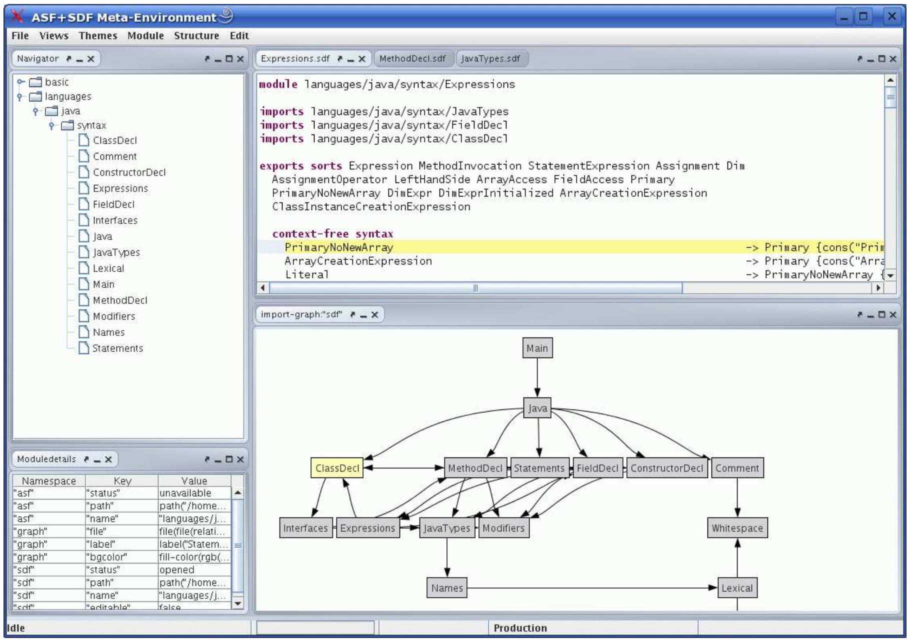

Figure 6: User interface of the Meta-Environment

4.2.3 PARSIFAL

The PARSIFAL[6] project works on proofs of programs and protocols and emphasizes the underlying principles of proof search. Typical applications are in areas like proof-carrying code and model checkers. The vice-head of PARSIFAL Joëlle Despeyroux was on the GIPE team.

4.2.4 TROPICS

The TROPICS[7] team works on an environment for analysis and transformation of scientific programs, and aims at applying techniques from software engineering to numeric software for Computational Fluid Dynamics. Their Tapenade[8] system applies Automatic Differentiation to Fortran programs in order to derive optimized versions of the original program. TROPICS is headed by Laurent Hascoët, who was a member of the GIPE team.

4.3 Ariane V

On June 4, 1996 the first test flight of the Ariane 5 (flight 501) took place and was a dramatic failure. After 37 seconds the rocket exploded.

According to the report of the Inquiry Board [62] the Ariane 5 reused software from Ariane 4 beyond its specifications and this caused a floating point conversion to fail. For reasons of efficiency the Ada error handler had been disabled. This has become known as one of the more expensive software bugs.

Gilles, being an expert on programming language semantics in general and on Ada semantics in particular, was a member of the Inquiry Board. Is there a better example of theory meeting practice?

[6] Preuves Automatiques et Raisonnement sur des SpécIFicAtions Logiques, see `http://www.lix.polytechnique.fr/parsifal/`.

[7] Program transformations for scientific computing, see `http://www.inria.fr/recherche/equipes/tropics.en.html`.

[8] See ftp://ftp-sop.inria.fr/tropics/tapenade/README.html.

4.4 Evaluation Committees and Scientific Council

In 1998 I participated in the evaluation committee for INRIA Programme 2A, in 1999 for the project OASIS and in the period 1998–2002 I served as member of INRIA's Scientific Council. All these occasions gave me ample opportunity to watch Gilles at work: friendly, hospitable and seemingly bored that yet another evaluation had to take place. At the same time he was very keen that the politically most desirable conclusions ended up in the final reports of these committees.

In the Scientific Council, Gilles acted as an excellent strategist with a keen eye for new scientific developments and for opportunities for INRIA to make a scientific or societal contribution. We shared an interest in the phenomenon of *spinoff companies*: attempts to bring scientific results to the market.

4.5 The Dutch side

On the Dutch side, we have seen slow progress in three generations of software:

1992 The initial LeLisp-based version of the ASF+SDF Meta-Environment.

2000 The first generation of a completely component-based Meta-Environment based on the ToolBus. The main implementation languages used were C, ASF+SDF and Tcl/TK (for the user-interface).

2007 The second generation, released in 2007, that contains a plugin architecture for the user-interface, visualization tools, and more, see Section 4.5.1 below and [16, 15, 63]. In this edition, Java has become one of the prominent implementation languages.

In the remainder of this section, I will give more details about this second generation system (The Meta-Environment version 2.0) as we present it today.

4.5.1 The Meta-Environment, Version 2.0

The Meta-Environment[9] is an open framework for language development, source code analysis and source code transformation. It consists of syntax analysis tools, semantic analysis and transformation tools, and an interactive development environment (see Figure 6). It is supported by a growing open source community, and can easily be modified or extended with third party components.

The Meta-Environment is a generalization of the ASF+SDF Meta-Environment that has been successfully used in many analysis, transformation and renovation projects. The Meta-Environment has been used for applications such as:

- Parsing (new and old) programming languages, for further processing the parse trees.
- Analysis of source code (fact extraction, type analysis, and documentation generation).
- Transformation, refactoring, and source code generation.
- Design and implementation of Domain-specific languages.

4.5.2 Features of The Meta-Environment

From the background given in this paper, the features of The Meta-Environment can be easily recognized:

- Modular grammar definitions—a consequence of our generalized parsers.
- Declarative disambiguation filters used to resolve many common ambiguities in programming languages.
- Conditional term rewriting used to perform software transformations.

[9]This section is based on a tool demonstration presented in [15]. See `www.meta-environment.org` for further details.

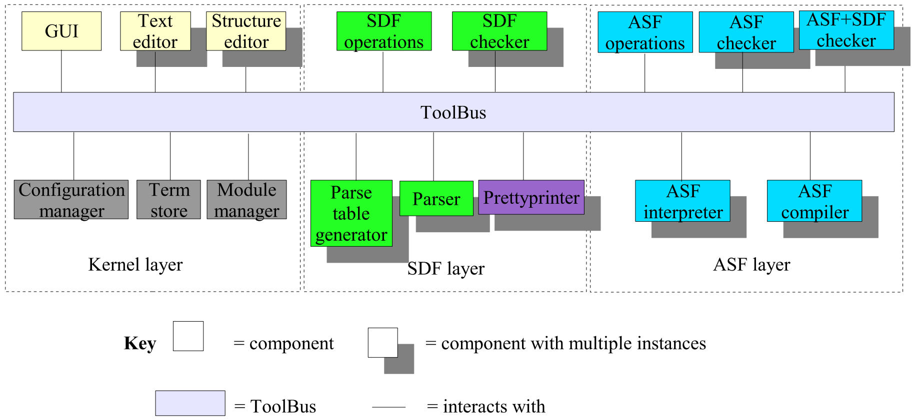

Figure 7: Run-time architecture of the Meta-Environment

- Seamless integration of user-defined syntax in rewrite rules, enabling the definition of transformation rules in concrete syntax, as opposed to using abstract syntax and getting much more obscure rules. This also guarantees fully syntax-safe source code generation.
- A highly modular and extensible architecture based on the coordination of language processing tools.
- ATerms as a language-independent intermediate data exchange format between tools.
- An Integrated Development Environment (IDE) that provides interactive support and on demand tool execution.

Version 2.0 of The Meta-Environment includes various new features:

- A grammar library containing grammars for C, Java, Cobol and other programming languages.
- Rewriting with layout. This enables fine-grained analysis and transformations such as high-fidelity source-to-source transformations that preserve comments and whitespace.
- Automatically generated syntax highlighting based on syntax definitions for arbitrary languages.
- Automatically generated prettyprinters that can be refined by the user.
- Rscript—a relational calculus engine that enables easy computing with facts extracted from source code.
- Advanced information visualization tools for the interactive display of relations, parse trees and dependencies.
- A fully customizable, plugin-based user-interface with dockable panes, and user-defined menus and buttons. Plugins can run user-defined scripts to interact with other tools.

A major architectural improvement in version 2.0 is the division of the system into several separate layers that enable the creation of a family of related applications that share common facilities such as user-interface, parsing infrastructure, and error reporting (the kernel layer). The facilities for syntax analysis (SDF layer) and transformation (ASF layer) are implemented on top of this kernel. See Figure 7 for an overview of this layered architecture. Observe that the system uses the ToolBus as coordination infrastructure and compare this with the Sophtalk approach (Section 3.4) and the architecture of SmartTools (Figure 5). All three systems achieve component decoupling by way of messaging middleware.

4.5.3 Applications of The Meta-Environment

In the area of software evolution, The Meta-Environment has been successfully applied to the transformation of database schemas, analysis of embedded SQL, Cobol prettyprinting and restructuring, PL/I parsing, analysis and restructuring of C++, dead-code detection in Java, and aspect mining in C.

Due to the many extension points (rules for defining syntax, prettyprinting, analysis and transformation; extensible user-interface; connection of third-party components; extensible architecture) the system can be easily adapted to the requirements of a specific software evolution or renovation problem.

In the area of domain-specific languages, the system has been applied to domains as disparate as financial systems and machine tooling.

4.6 Synthesis: the ATEAMS project

As in every classical story there are three parts: thesis, anti-thesis and synthesis. The GIPE story also seems to follow this structure: the GIPE projects (see Stcion 2) and the post-GIPE era (see Section 4) form thesis and anti-thesis. The synthesis suddenly enters the stage by way of a joint CWI/INRIA project team that is in preparation at the time of writing: ATEAMS[10] that will do research on *fact extraction*, *refactoring and transformation*, and *reliable middleware* with as overall aims to enable the evolution of large software systems to service-oriented systems and to use the service paradigm to scale up analysis and transformation tools.

5 Concluding remarks

> **Gilles the Meta-technologist.** *Gilles would often exclaim: "This is so meta" with a strong emphasis on the second syllable of the word "meta".*

Meta-approaches were, are and will remain crucial: meta-modeling, model-driven development, and domain-specific engineering are the current labels for the activities that played a prominent role in the objectives and results of the GIPE projects.

It comes as no surprise that generic language technology is in increasing demand for the design and implementation of domain-specific languages; for the analysis and transformation of software and software models; and for numerous forms of code generation. The increasing popularity of the Meta-Environment is one illustration of this.

As this brief historical overview shows, many of the ideas Gilles worked on are still in daily use today. This is mostly due to his conceptual approach to many problems. He liked to view things from a higher level of abstraction. First in studying meta-level descriptions of programming languages, later as scientific director of INRIA where he could supervise and steer technological developments and research directions. A true "meta-technologist" with a vision. As president of INRIA he could apply his meta-skills and vision to a large, bureaucratic but highly successful research organization. Gilles was the opposite of a bureaucrat, but he knew as no other that conquering the bureaucracy is the only way to realize one's vision.

References

[1] I. Attali. A natural semantics for Eiffel dynamic binding. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(5), 1996.

[2] I. Attali, D. Caromel, and A. L. Wendelborn. From a formal dynamic semantics of Sisal to a Sisal environment. In *HICSS (2)*, pages 266–267, 1995.

[10] Analysis and transformation of EvolvAble Modules and Services.

[3] I. Attali, J. Chazarain, and S. Gilette. Incremental evaluation of natural semantics specification. In M. Bruynooghe and M. Wirsing, editors, *PLILP*, volume 631 of *Lecture Notes in Computer Science*, pages 87–99. Springer, 1992.

[4] I. Attali, C. Courbis, P. Degenne, A. Fau, J. Fillon, Chr. Held, D. Parigot, and C. Pasquie. Aspect and XML-oriented semantic framework generator: Smarttools. In *Second Workshop on Language Descriptions, Tools and Applications, LDTA'02*, volume 65 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 1–20. Springer, 2002.

[5] I. Attali, C. Courbis, P. Degenne, A. Fau, D. Parigot, and C. Pasquier. Smarttools: a generator of interactive environment tools. *Electr. Notes Theor. Comput. Sci.*, 44(2), 2001.

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

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

[8] J.A. Bergstra and P. Klint. The discrete time ToolBus – a software coordination architecture. *Science of Computer Programming*, 31(2-3):205–229, July 1998.

[9] Y. Bertot. Implementation of an interpreter for a parallel language in Centaur. In *European Symposium on Programming*, pages 57–69, 1990.

[10] Y. Bertot. Occurrences in debugger specifications. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 327–337, New York, NY, USA, 1991. ACM Press.

[11] Y. Bertot. Origin functions in lambda-calculus and term rewriting systems. In J.-C. Raoult, editor, *CAAP*, volume 581 of *Lecture Notes in Computer Science*, pages 49–65. Springer, 1992.

[12] Y. Bertot, G. Kahn, and L. Théry. Proof by pointing. In M. Hagiya and J. C. Mitchell, editors, *TACS*, volume 789 of *Lecture Notes in Computer Science*, pages 141–160. Springer, 1994.

[13] S. Billot and B. Lang. The structure of shared forests in ambiguous parsing. In *ACL*, pages 143–151, 1989.

[14] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: The system. In *Software Development Environments (SDE)*, pages 14–24, 1988.

[15] M.G.J. van den Brand, M. Bruntink, G.R. Economopoulos, H.A. de Jong, P. Klint, T. Kooiker, T. van der Storm, and J.J. Vinju. Using the meta-environment for maintenance and renovation. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 331–332. IEEE Computer Society, March 21-23 2007.

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

[17] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30:259–291, 2000.

[18] M.G.J. van den Brand and P. Klint. ATerms for manipulation and exchange of structured data: It's all about sharing. *Information and Software Technology*, 49(1):55–64, 2007.

[19] M.G.J. van den Brand, A.T. Kooiker, J.J. Vinju, and N.P. Veerman. A Language Independent Framework for Context-sensitive Formatting. In *10th Conference on Software Maintenance and Reengineering (CSMR 2006)*, pages 631–634. IEEE Computer Society Press, 2006.

[20] M.G.J. van den Brand, A. Sellink, and C. Verhoef. Control flow normalization for COBOL/CICS legacy system. In *CSMR*, pages 11–20. IEEE Computer Society, 1998.

[21] M.G.J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Programming Languages and Systems*, 5(1):1–41, 1996.

[22] Centaur web pages, Last visit December 2006. `http://www-sop.inria.fr/croap/centaur/centaur.html`.

[23] J. Chailloux, M. Devin, F. Dupont, J.-M. Hullot, B. Serpette, and J. Vuillemin. Le_Lisp version 15.2, le manuel de réf'erence. Technical report, INRIA, 1986.

[24] D. Clément. The natural dynamic semantics of Mini-Standard ML. In H. Ehrig, R. A. Kowalski, G. Levi, and U. Montanari, editors, *TAPSOFT, Vol.2*, volume 250 of *Lecture Notes in Computer Science*, pages 67–81. Springer, 1987.

[25] D. Clément. A distributed architecture for programming environments. In R.N. Taylor, editor, *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*, pages 11–21, 1990.

[26] D. Clément, J. Despeyroux, Th. Despeyroux, L. Hascoet, and G. Kahn. Natural semantics on the computer. Technical Report RR416, I.N.R.I.A., june 1985.

[27] D. Clément, J. Despeyroux, Th. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. In *LISP and Functional Programming*, pages 13–27, 1986.

[28] D. Clément, J. Heering, J. Incerpi, G. Kahn, P. Klint, B. Lang, and V. Pascual. Preliminary design of an environment generator. Second annual review report: D9, GIPE, ESPRIT Project 348, January 1987.

[29] D. Clément, J. Incerpi, and G. Kahn. CENTAUR: Towards a ”software tool box” for programming environments. In F. Long, editor, *SEE*, volume 467 of *Lecture Notes in Computer Science*, pages 287–304. Springer, 1989.

[30] D. Clément, V. Prunet, and F. Montagnac. Integrated software components: A paradigm for control integration. In A. Endres and H. Weber, editors, *Software Development Environments and CASE Technology*, volume 509 of *Lecture Notes in Computer Science*, pages 167–177. Springer, 1991.

[31] Y. Coscoy, G. Kahn, and L. Théry. Extracting text from proofs. In M. Dezani-Ciancaglini and G. D. Plotkin, editors, *TLCA*, volume 902 of *Lecture Notes in Computer Science*, pages 109–123. Springer, 1995.

[32] J. Coutaz. The box, a layout abstraction for user interface toolkits. Technical Report CMU-CS-84-167, Carnegie Mellon University, 1984.

[33] Th. Despeyroux. Executable specification of static semantics. In G. Kahn, D. B. MacQueen, and G. D. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 215–233. Springer, 1984.

[34] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.

[35] A. van Deursen, P. Klint, and F. Tip. Origin tracking. *J. Symb. Comput.*, 15(5-6):523–545, 1993.

[36] V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, and J.J. Lévy. A structure oriented program editor: a first step towards computer assisted programming. In *International Computing Symposium*. North Holland, 1975.

[37] V. Donzeau-Gouge, G. Kahn, and B. Lang. On the formal definition of ADA. In N. D. Jones, editor, *Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*, pages 475–489. Springer, 1980.

[38] V. Donzeau-Gouge, G. Kahn, B. Lang, and B. Mélèse. Documents structure and modularity in mentor. In *Software Development Environments (SDE)*, pages 141–148, 1984.

[39] V. Donzeau-Gouge, G. Kahn, B. Lang, B. Mélèse, and E. Morcos. Outline of a tool for document manipulation. In *IFIP Congress*, pages 615–620, 1983.

[40] V. Donzeau-Gouge, B. Lang, and B. Mélèse. Practical applications of a syntax directed program manipulation environment. In *ICSE*, pages 346–357, 1984.

[41] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

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

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

[44] J. Heering and P. Klint. Prehistory of the ASF+SDF system (1980–1984). In M.G.J. van den Brand, A. van Deursen, T.B. Dinesh, J. Kamperman, and E. Visser, editors, *Proceedings of ASF+SDF95 A workshop on Generating Tools from Algebraic Specifications*, number P9504 in Technical Report. Programming Research Group, University of Amsterdam, 1995.

[45] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *IEEE Transactions on Software Engineering*, 16(12):1344–1350, 1990.

[46] I. Jacobs, F. Montignac, J. Bertot, D. Clément, and V. Prunet. The Sophtalk reference manual. Technical Report 149, INRIA, February 1993.

[47] G. Kahn. An approach to system correctness. In *SOSP*, pages 86–94, 1971.

[48] G. Kahn, editor. *Semantics of Concurrent Computation, Proceedings of the International Sympoisum, Evian, France, July 2-4, 1979*, volume 70 of *Lecture Notes in Computer Science*. Springer, 1979.

[49] G. Kahn. Natural semantics. In F.-J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *STACS*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.

[50] G. Kahn, B. Lang, B. Mélèse, and E. Morcos. Metal: A formalism to specify formalisms. *Science of Computer Programming*, 3(2):151–188, 1983.

[51] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In *IFIP Congress*, pages 993–998, 1977.

[52] P. Klint. An overview of the summer programming language. In *Conference Record of the 7th ACM Symposium on Principles of Programming Languages (POPL'80*, pages 47–55, 1980.

[53] P. Klint. Formal language definitions can be made practical,. In *Algorithmic Languages*, pages 115–132, 1981.

[54] P. Klint. A survey of three language-independent programming environments. RR 257, INRIA, 1983.

[55] P. Klint. *A Study in String Processing Languages*, volume 205 of *LNCS*. Springer-Verlag, 1985. Based on the dissertation *From Spring to Summer*, defended at the Technical University Eindhoven, 1982.

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

[57] 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, University of Amsterdam, 1993.

[58] H. Korte, H. Joosten, V. Tijsse, A. Wammes, J. Wester, Th. Kuhne, and Chr. Thies. Design of a LOTOS simulator: Centaur from a user's perspective. Fifth annual review report: D5, GIPE II, ESPRIT project 2177, 1993.

[59] B. Lang. Deterministic techniques for efficient non-deterministic parsers. In J. Loeckx, editor, *ICALP*, volume 14 of *Lecture Notes in Computer Science*, pages 255–269. Springer, 1974.

[60] B. Lang. On the usefulness of syntax directed editors. In R. Conradi, T. Didriksen, and D. H. Wanvik, editors, *Advanced Programming Environments*, volume 244 of *Lecture Notes in Computer Science*, pages 47–51. Springer, 1986.

[61] B. Lang. The virtual tree processor. In J. Heering J. Sidi and A. Verhoog, editors, *Generation of Interactive Programming Environments, Intermediate Report*, number CS-R8620 in Technical Report. Centrum voor Wiskunde en Informatica, 1986.

[62] J.L. Lions. ARIANE 5: Flight 501 Failure, Report by the Inquiry Board. `http://homepages.inf.ed.ac.uk/perdita/Book/ariane5rep.html`, 1996. Last visit January 2007.

[63] Meta-Environment web pages, Last visit March 2008. `http://www.meta-environment.org`.

[64] E. van der Meulen. Deriving incremental implementations from algebraic specifications. In M. Nivat, Ch. Rattray, T. Rus, and G. Scollo, editors, *AMAST*, Workshops in Computing, pages 277–286. Springer, 1991.

[65] E. Morcos-Chounet and A. Conchon. PPML: a general purpose formalism to specify prettyprinting. In H.-J. Kugler, editor, *Information Processing 86*, pages 583–590. Elsevier, 1986.

[66] P.A. Olivier. Debugging distributed applications using a coordination architecture. In D. Garlan and D. Le Métayer, editors, *COORDINATION*, volume 1282 of *Lecture Notes in Computer Science*, pages 98–114. Springer, 1997.

[67] D.C. Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems*, 2(4):465–483, 1980.

[68] Sophtalk web pages, Last visit December 2006. `http://www-sop.inria.fr/croap/sophtalk/sophtalk.html`.

[69] L. Théry, Y. Bertot, and G. Kahn. Real theorem provers deserve real user-interfaces. In *SDE 5: Proceedings of the fifth ACM SIGSOFT symposium on Software development environments*, pages 120–129, New York, NY, USA, 1992. ACM Press.

[70] M. Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1985.