Constructing SDEs with the IPSEN Meta Environment

Peter Klein and Andy Schürr
Department of Computer Science III, Aachen University of Technology, Germany
(pk|andy)@i3.informatik.rwth–aachen.de

Abstract

We describe a high–level approach to the construction of software development environments (SDEs) featuring a new degree of automation. It combines a variety of reuse approaches into one powerful machinery. Based on suitable specification formalisms (context–free grammars as well as graph rewriting systems), generator tools, and a framework implementation, the IPSEN meta environment allows the construction of software development environments supporting textual and graphical editors, analyzers, and other complex tools. The meta environment itself has also been bootstrapped from specifications and the framework.

1. Introduction

An essential aim in the development of software systems in general and software development environments in particular has always been to *automate* as much of the *construction process* as possible. Reuse on different levels (component/library code, design of frameworks and patterns, generators) has been identified as a major contribution to progress in this area.

Over the last ten years, our group has intensively studied the area of the construction of *integrated, interactive, and incremental software development environments* in the IPSEN project. With growing experience, we have reached a high level of automation in this process: Based on graphs as the fundamental data structure, we started with a graph–oriented database and a handcoded environment. Then, we gradually isolated general and generic parts in the framework implementation and created an environment to generate context–free editors from EBNFs. Furthermore, we used graph rewriting rules to specify the behavior of tools and translated the rules manually into Modula–2 code. After we could automatically generate code from the graph grammar rules, we decomposed the specifications themselves into reusable and specific parts. As a result, we are now reusing not only design and code, but also specifications and generator tools wherever possible.

The overall results of the project are described in great detail in [16]. This paper is a condensed and newly focused sketch of the generator machinery used in IPSEN. Section 2 gives a short discussion of comparable SDE generating approaches. In section 3, we will give an overview of the generator machinery. Then, we will discuss the specification languages we use in the meta environment. Section 5 sketches the architecture of the IPSEN framework and section 6 describes the generated components and their interaction with the framework. A short summary concludes this paper.

2. Related Work

Generating *complete software engineering environments* is a rather ambitious goal which will not be reached in the nearby future. The first generation of research projects like Mentor [5], PSG [1, 21], or CPSG [18] had a main focus on generating language–sensitive programming environments for text–oriented languages. They proposed a *single formalism* like attribute grammars for specification purposes. Their main disadvantage was that generated environments and the generation process itself were *not open for manual modifications* or extensions. Therefore, these approaches failed as soon as certain tool aspects could not be described in the formalism offered.

Later on, more flexible systems like Centaur [3] were developed which provide a *variety of formalisms* tailored to specific purposes and generate components of an *open architecture*. Another tendency, apart from the employment of special–purpose tool generation languages, was to complement attribute grammars with *more general–purpose formalisms* like relational algebras [8], horn–clause logic [2], or algebraic specifications [11]. This simplified the specification of complex static semantics rules or a language's dynamic semantics considerably.

But all approaches mentioned above were and are mainly useful for *text–oriented languages* which have a dominant (abstract syntax) tree skeleton as their underlying logical structure. This is in sharp contrast to the observation that a majority of commercial CASE tools are syntax–directed diagram editors or, more general, *diagram (visual language) processing tools*. Therefore, new techniques – beyond the usage of graphics libraries, user interface toolkits, or generic diagram editors – are urgently needed for generating editors, analyzers, debuggers, and the like for visual languages.

Published in
Proceedings of the 8th Conference on Software Engineering Environments SEE'97,
Los Alamitos: IEEE Computer Society Press (1997), pp. 2–10

Currently existing *visual language definition formalisms*, like picture layout grammars [7] or context–free hyper graph grammars [14], and their accompanying generators are still rather immature compared to attribute grammar–based approaches. Resulting specifications tend to become very large and unmaintainable due to the lack of module or abstraction concepts as they were developed for attribute grammars [9, 17].

Finally, we are not aware of any syntax definition formalisms which are well–suited for the specification of *hybrid languages*, i.e. for languages with nested textual and visual sublanguages or for languages which have both a textual and a visual representation. As a consequence, it is still very difficult to mechanize the development of software engineering tools as presented in fig. 1 of section 3.

3. The IPSEN Meta Environment

In the following, we will present our approach for the *development* of integrated software development environments. It has the following *characteristics*:

- Context–free grammars (EBNF) are used to describe the syntax and the layout of languages with a textual representation.
- Graphical layout annotations and programmable views may be used to define additional diagrammatic (sub–) representations.
- Programmed graph rewriting systems (PROGRES) are our formalism for all logical aspects which are outside the scope of EBNFs.
- All tool generators have the form of UNIX filters; they consume source code templates as input and produce human–readable source code instances as output.
- Generated language–specific pieces of code play well–defined roles as components of an environment framework which implements the overall (generic) behavior of tools.
- The IPSEN meta environment for editing EBNF and PROGRES specifications is generated by means of itself.
- Almost all generic environment components are handcoded, although some of them have generic PROGRES specifications.
- Handcoding is also a frequently used strategy for replacing time–critical pieces of generated code specified in PROGRES.

It is impossible to discuss all these aspects within a single paper. Therefore, we will focus our interest onto those three aspects which are – in our opinion – the most important ones of the IPSEN project: (1) the combined usage of *EBNFs and graph rewriting systems* (section 4), the prominent role of a *framework architecture* with well–defined (open) interfaces (section 5), and (3) the UNIX like approach to implement *tool generators as text filters* (section 6). For any further details, the reader is referred to [16].

We are now going to describe how instances of IPSEN environments are built using the *IPSEN meta environment*. As an example, we want to show the development of a technical environment for Requirements Engineering and Programming–in–the–Large as depicted in the screen dump of fig. 1. In a first step, this environment should contain an editor for a module interconnection language (MIL) with a coarse–grained graphical representation and a fine–grained textual representation. Accordingly, the task of the meta environment user is to specify the language the editor should support. This specification comprises two parts for the context–free and the context–sensitive properties of the language, respectively.

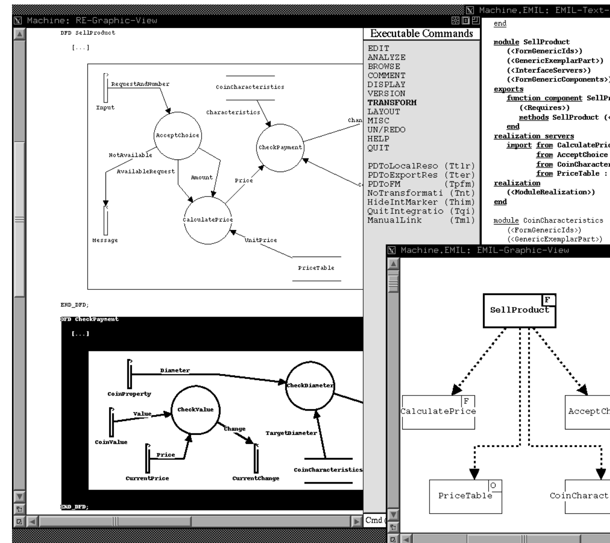


Figure 1. Screen dump of an IPSEN environment

To provide a base for the context–sensitive properties, the process starts with the definition of the language's *concrete and abstract syntax* in the form of *EBNF rules*. These EBNF rules are used to generate the core functionality of a syntax–directed MIL editor with text–only document representations. The layout of a rule in the EBNF is also used as a template for the layout of the corresponding construct in the document representation.

Additional EBNF rule annotations may be used to specify graphical MIL language representations and to generate *graphical editors*. Essentially, the respective annotations assign representation attributes concerning the graphical structure (being a vertex, an edge, a half–edge etc.) to EBNF nonterminals. A separate table adds concrete presentation information (a box with a certain line style etc.) to the corresponding increment types. It should be noted that annotated EBNFs have been used successfully even for languages which seem to be purely graphical (like the DFDs above). The advantages of modeling a language in terms of abstract syntax, like the possibility to derive textual representations for arbitrary documents and therefore to store or exchange them as human–readable ASCII

files, generally make up for the necessity to devise an EBNF for a language. Nevertheless, in the worst case, this approach is rather awkward if the EBNF structure does not reflect the logical structure of the document appropriately. Therefore, we are currently working on an extension of the machinery presented here which allows the specification of a visual language's syntax directly based on graphs and graph rewriting rules (graph grammars) [19].

The diagram of fig. 2, especially its subdiagram A, shows how a *context–free editing environment is generated* from these sources of input. An EBNF editor (box 1) is used to define a language's context–free syntax (including all above mentioned annotations). The EBNF generator (box 4) then takes a complete EBNF document (box 2) plus a number of text file skeletons (box 3) as input. It generates all language–specific Modula–2 sources (box 5) which are needed to realize syntax–directed and free editing of MIL documents via mixed graphical/text–oriented user interfaces. The EBNF environment (editor and generator) itself is an IPSEN environment constructed from a bootstrapping step.

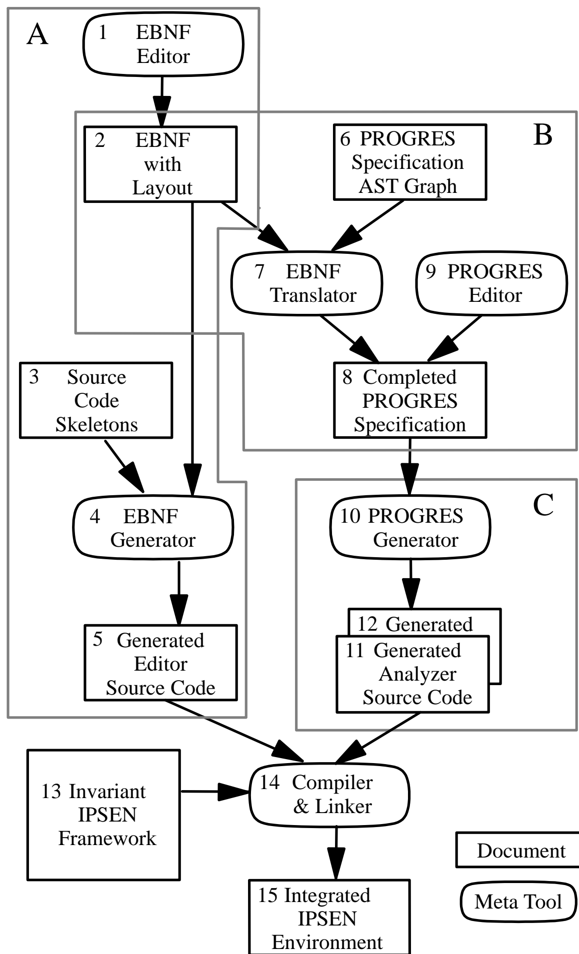


Figure 2. The IPSEN meta environment

The generated sources together with the invariant IPSEN framework (box 13) are compiled and linked (box 14) to construct a *basic version of the environment* (box 15). This environment does not yet offer any analysis or complex document transformation commands, e.g. for preventing cyclic import relationships or producing configurations with certain properties. Its overall structure and most important basic components will be discussed in section 5.

The next step comprises the specification of *context–sensitive document processing operations* for any kind of analysis or execution tools by means of *programmed graph rewriting systems*. The corresponding rules are specified in the PROGRES language which will be sketched in section 4 below.

To use the context–free EBNF specification as a framework for context–sensitive operations, an initial PROGRES document specification is systematically derived from the EBNF. This kind of modeling knowledge is currently hardwired into an incrementally working *EBNF translation tool* (box 7). The translator takes an EBNF (box 2) and a language–independent abstract syntax (AST) graph specification (box 6) as input and produces a corresponding language–specific PROGRES specification as output (box 8). It thereby forms the connection between the EBNF and PROGRES tools.

The generated initial specification must then be augmented with *context–sensitive graph transformations* (box 9). The PROGRES environment used for this purpose is, again, an IPSEN environment. The manually added graph transformations (but not the generated basic transformations for context–free editing) are translated into various pieces of Modula–2 code (boxes 11 and 12) by means of a code generation machinery (box 10).

Finally, all generated source code fragments are embedded into the invariant IPSEN architecture framework to construct a complete environment. The interactions between editing EBNF documents and generated PROGRES documents, the contents of subdiagram B of fig. 2, is explained in more detail in section 4. The last subdiagram C of fig. 2, the translation of PROGRES specifications into Modula–2 code, will not be explained in detail here.

4. EBNF & PROGRES as Input Languages

The previous section discussed the SDE generation process on a very coarse grained level. EBNF and PROGRES were mentioned as specification languages. The definition of a language's context–free syntax by means of Extended Backus Naur Forms (EBNF) is rather straight–forward and needs no further explanation except for the presentation of a cutout of the MIL language syntax definition. It is presented in fig. 3.a).

A MIL specification comprises a list of modules of different categories (abstract data type module, function module etc.) which are subject to different context–sensitive

constraints. Modules have an identifier (declaration) and a definition. The context–free structure of such a definition is omitted here, but we assume that it contains a list of exported resources (type and procedure declarations), a list of imported modules, and a list of implementation variants.

The MIL language's EBNF is sufficient to generate a context–free MIL editor with a text–oriented interface. Additional graphical layout annotations and view definition mechanisms are needed to generate a MIL diagram editor (cf. [16]). We have to switch from EBNFs to *programmed graph rewriting systems* (PROGRES specifications) as soon as context–sensitive aspects are regarded.

A PROGRES specification consists of two main parts: (1) a *graph schema* which determines the internal structure of the respective graph class and (2) a set of *graph transformations*. Static type checking rules guarantee that the graph transformations generate schema consistent graphs only.

a) EBNF of Module Interconnection Language (MIL) – cutout:

```
(1) System ::= "system" DeclSystemId ";"    (* Layout of EBNF = pretty printing rules *)
                   ModuleList
                "end" "."
(2) ModuleList ::= { Module } Module
(3) Module ::= ADTModule | FunctionModule | ...       (* 3 different types of modules *)
(4) ADTModule ::= "adt" DeclModuleId ";"        (* syntax of abstract data type module *)
                    ADTDef     (* ADTDef = export interface with one type declaration *)
                  "end" ";"     (* + list of implementation variants (not shown here) *)
```

b) Generic graph schema for abstract syntax graphs – cutout:

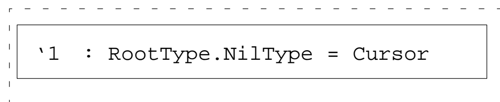
```
node class BINARY_OP is a INCREMENT (* abstract node class without node instances *)
  meta
    1stType : type in INCREMENT;    (* determines type of first child node  *)
    2ndType : type in INCREMENT;    (* determines type of second child node *)
    NilType : type in PLACEHOLDER;  (* placeholder for binary subtree       *)
end;
edge type 1st : BINARY_OP -> INCREMENT;
edge type 2nd : BINARY_OP -> INCREMENT;
```

c) Generated specific graph schema for MIL graphs – cutout:

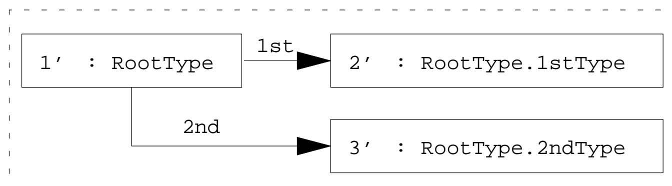
```
node class MODULE is a INCREMENT end;  (* abstract node class without node instances *)
node type ADTModule is a MODULE, BINARY_OP  (* node type = concrete class with nodes *)
  redef meta
    1stType := DeclModuleId;     (* DeclModuleId = type of first MIL graph child     *)
    2ndType := ADTDef;           (* ADTDef = type of second MIL graph child         *)
    NilType := NilModule;        (* placeholder for unexpanded module in module list *)
 constraint  (* manually added constraint: uses path down to compute all type decls. *)
    card(self.downTo(TypeDecl)) = 1;    (* in ADTDef, no. of type decls. must be 1  *)
end;
```

d) Example for generic abstract syntax graph modifying operation:

```
production CreateBinary(     Cursor : PLACEHOLDER ; RootType : type in BINARY_OP;
                         out NewCursor : RootType) =
```




```
(* Cursor points to unexpanded
   subtree. The type of the placeholder
   must be the placeholder type for
   the RootType of the new subtree.  *)
::=
```



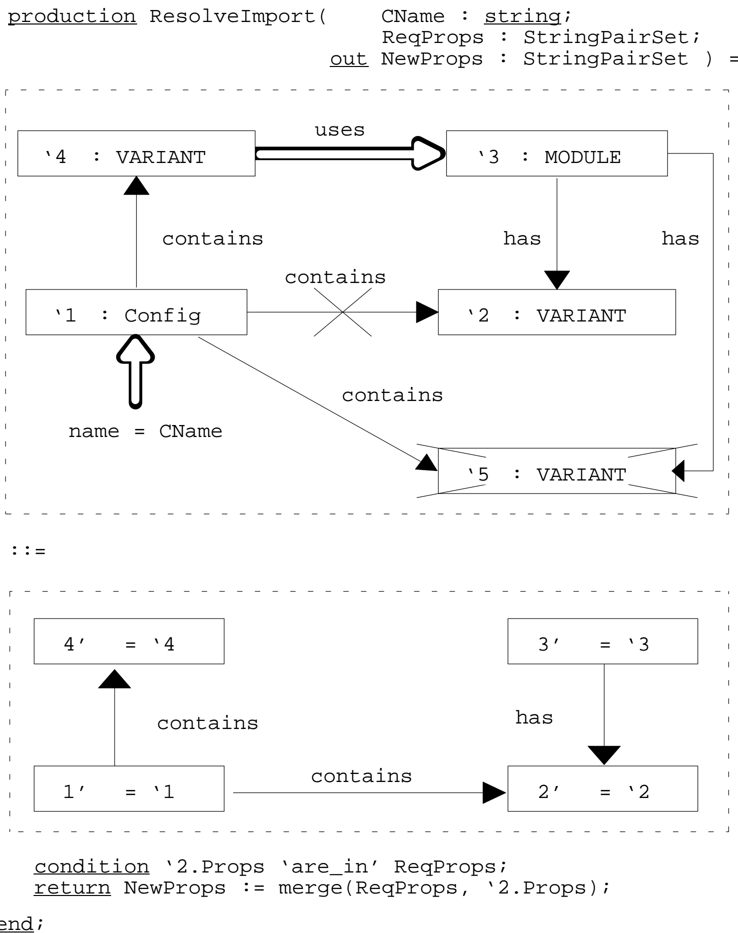

```
(* The meta attributes
   1stType and 2ndType
   of RootType determine
   the types of children
   nodes. The old node is
   deleted and three nodes
   are created. The new
   subtree root is returned
   in NewCursor.            *)

  embedding                           (* redirect incoming tree edges from  *)
      redirect <-1st-, ... from ‘1 to 1’;   (* placeholder to root of new subtree *)
  return NewCursor := 1’;      (* assigns root id of new subtree to out parameter *)
end;
```

Figure 3. Cutout of the generic abstract syntax graph specification

As already mentioned, a first version of a PROGRES specification is generated from the MIL EBNF. It defines the internal structure of abstract MIL syntax graphs and context–free editing operations. Using subtype and parametric polymorphism, we were able to split such a specification into three parts. The first one is a *language–independent graph schema* which defines the overall structure of abstract syntax graphs and corresponding operations. Fig. 3.b) shows one of its node classes. `BINARY_OP` is an abstract node class with three class (meta) attributes. These attributes have to be instantiated in any MIL language–specific (concrete) node class declaration which has `BINARY_OP` as one of its superclasses. This guarantees the context–free correctness of abstract syntax graphs.

Fig. 3.c) is a cutout of the second part of a generated specification. It contains two *MIL specific node classes* which correspond to the EBNF rules number (3) and (4). The first one defines the class of all types of modules, the second one the class of abstract data type modules. Its superclass `BINARY_OP` defines its structural properties, the superclass `MODULE` may be used to declare properties which all types of modules have in common. The redefinitions of its inherited meta attributes are generated, whereas the constraint was added manually. It requires that an ADT module exports exactly one type declaration. It is one example of how *static semantics rules* may be defined in PROGRES.

The last (third) part of the generated specification is a *set of graph transformations* for context–free editing purposes. Fig. 3.d) displays one example of this kind, a parameterized production which replaces a given placeholder node by one of its legal expansion alternatives (with two syntax tree children). Its input parameter `Cursor` corresponds to the highlighted MIL placeholder at the user interface. Its output parameter `NewCursor` is used to highlight the expansion result at the user interface or to pass the created subtree (root) to a subsequent graph transformation as input. The last input parameter `RootType` is the type (label) of the new subtree root node.




```
(* First, selects a variant '4
   which is part of the already
   built configuration with the
   required name CName (expres-
   sion below double arrow at
   node '1).
   It then extends its match to
   a module '3 which is used by
   the selected variant. The
   double arrow uses denotes a
   separately defined path
   through the graph from a
   module's variant to all of
   its imports.
   Finally, a variant '2 is
   selected which has the
   required properties ReqProps
   (condition at bottom of
   fig.) and is not part of the
   configuration (crossed-out
   contains edge).
   It is an additional require-
   ment that no other variant
   of module '3 is already part
   of the current configuration
   (crossed-out node '5).
   The production preserves all
   matched nodes and edges
   (node inscriptions n' = 'n).
   It creates a single contains
   edge from
   the configuration 1' to the
   selected variant  2'.
   It returns a new set of
   configuration properties
   NewProps which is the result
   of combining the old set of
   properties ReqProps and the
   new variant's properties. *)
```

Figure 4. Example for a complex language–specific graph transformation

The expression `RootType.NilType` in the production's left–hand side (above ::=) guarantees that a call of `CreateBinary` with `RootType = ADTModule` fails if the type of its input parameter `Cursor` is not equal to type `NilModule`. The other two meta attribute referencing expressions, `RootType.1stType` and `RootType.2ndType`, in the production's right–hand side (below ::=) are responsible for generating child nodes with context–free correct types. This is our approach to specify generic *language–independent productions* which nevertheless preserve language–specific properties.

Fig. 4 contains one example of a more complex *MIL–specific graph transformation*. It may be used to extend a partially constructed software system configuration step by step such that all selected implementation variants fulfill certain properties. For further details concerning the specification language PROGRES, our graph grammar engineering methodology, and the selected MIL example, the reader is referred to [6, 16, 20].

5. The IPSEN Framework Architecture

As was already mentioned, the code generated from the different input sources is embedded into a framework architecture, cf. box 13 in fig. 2. This framework is invariant with respect to the languages and tools the system eventually supports and contains general components for user interface handling and the like. To get a better understanding of what is actually generated and how the generated parts cooperate with the general framework parts, this section explains the architecture on a coarse–grained level, cf. fig. 5. We will not go into the details of the architecture description language used here as this would be beyond the scope of this paper; please refer to [15] for a detailed description.

The *bottom layer* of the IPSEN framework contains data abstraction components which hide the details of the underlying infrastructure, e.g. operating system, window system, and the like. It contains the following components:

GRAS: The GRAph Storage subsystem encapsulates a graph–based non–standard database system specifically designed for (software) engineering environments [10].

IPSEN Window System: This subsystem handles the mapping of an abstract, system–independent window system onto a concrete realization, possibly with the help of an existing window system.

The next level of the framework provides application–specific *logical document* classes and additional structures serving the *interaction* between user and application.

Logical Document: Logical documents contain the actual information the user is working with. Every logical document is an instance of a document class (an abstract data type) which is based on a certain language.

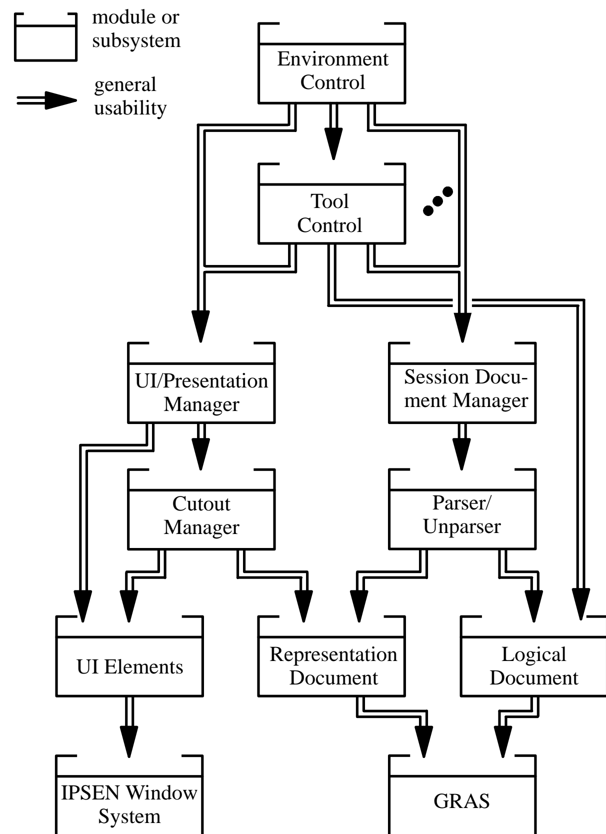


Figure 5. Coarse architecture of an IPSEN env.

Representation Document: As logical documents contain no information about how to visualize their contents in a human–readable way, separate documents are needed to store the representation structure of a logical document. Like logical documents, representation documents are stored persistently in the GRAS database.

UI Elements: This subsystem encapsulates all services necessary for the communication between the system and the user. Noticeably, it provides the views (called presentations) a user has on the document he is working with.

The next architecture layer facilitates *updates* between a logical document, its representation documents, and its presentations. All transformators work *incrementally*, i.e. they update only the portion of the target structure which directly relates to the changed part in the source structure.

Parser/Unparser: The unparser propagates changes in the logical document into respective representation documents by inserting or deleting textual and graphical objects. Vice versa, the parser can infer the logical structure from a given textual representation.

Cutout Manager: The cutout manager is the corresponding subsystem for the coupling between representations and presentations. It determines which part of the document is visible to the user and creates screen objects corresponding to the objects in the representation graph with respect to zooming and other display parameters.

The fourth level of the framework comprises components which know about all *documents* and *presentations* and their *relationships*, i.e. which representation document depends on which logical document etc.

Session Document Manager: The session document manager is responsible for the coupling of logical and representation documents. It is in charge of access control to documents, keeping track of versioning information, and controlling incremental updates between logical and representation documents.

UI/Presentation Manager: This is the session document manager's counterpart for the coupling of representation documents and document presentations. It knows about a representation document's open presentations and their display parameters.

The next two layers consist of components for *controlling* one tool and all tools of an environment, respectively.

Tool Control: For every tool attached to an IPSEN environment, there is exactly one tool control subsystem. Depending on the type(s) of the underlying logical document(s), there are control modules e.g. for editor, analyzer, execution, monitoring, and integration tools. This subsystem is in charge of executing the commands selected by the user.

Environment Control: This top–level module manages the interaction between the user and the tools and is responsible for the consistency of logical, representation, and presentation documents.

6. The Tool Generation Process

Based on the architecture description above, we can now describe in more detail where the generated parts of an IPSEN instance fit into the framework.

From the overview in section 3 follows that the language–specific parts of an environment are generated from two different sources: the EBNF providing the context–free syntax and the PROGRES specification from which context–sensitive properties as well as complex tool functionality can be derived. Based on fig. 6, we describe the context–free part first.

As was already mentioned, two aspects of the EBNF specification are used by the generation machinery: The *logical document* containing the information about the structure of the language and an accompanying *representation document* used to derive the layout of specific language constructs in the editor. From these documents, several output files are generated. Fig. 6 particularly shows that the files are not created from scratch, instead, they are constructed by replacing placeholders in a skeleton file by appropriate values. In this sense, the activity of the EBNF generator resembles a generic instantiation of templates (the skeleton files) with language–specific parameters (derived from the EBNF). This procedure restricts the complexity of the generator to a minimum and allows experienced users to fine–tune the behavior of the generated output by changing the skeletons.

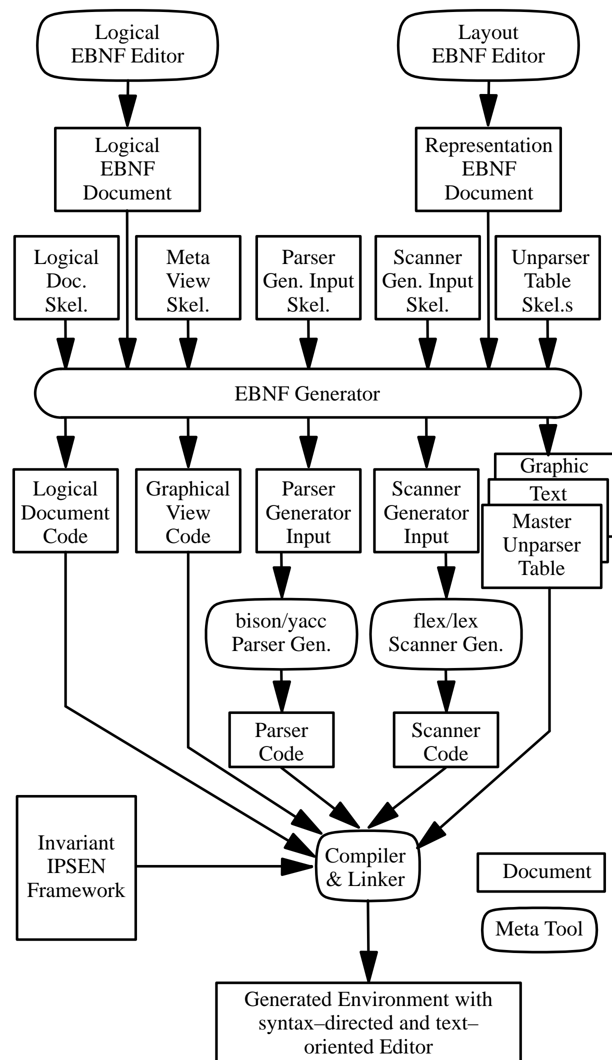


Figure 6. Generating context–free editors

We now give a short description of the *generated files* and their purpose with respect to the architecture discussion from section 5.

Logical Document Code: This part of the EBNF generator output is part of the logical document subsystem. Essentially, it maps language–specific abstract syntax tree (AST) operations onto the general graph–oriented interface of the GRAS database. This part of the EBNF generator output is only used for context–free editors; it may be replaced by code generated from the PROGRES specification later.

Unparser Tables: These tables, part of the parser/unparser subsystem in fig. 5, describe the mapping of logical structure to representation objects. The unparser actually consists of two separate components for the handling of textual and graphical language elements, each of them parameterized by the tables produced here. The master

unparser coordinates these components by looking up (in the master unparser table) whether a nested substructure should be represented textually or graphically, calling the corresponding unparser, and assembling the substructure representations into a representation for the overall structure.

Scanner/Parser Generator Input: Although the machinery presented here generally produces command–driven syntax–oriented editors, it is also possible for the user to enter or modify arbitrary textual increments of the document with a customary text editor. This procedure of "free input" is facilitated by unparsing the corresponding increment into textual form, passing it to the editor, and scanning and parsing the resulting text back into the logical document. As can be seen in fig. 6, the generator does not produce scanner and parser itself. Instead, input files for the well–known lex/flex and yacc/bison scanner/parser generators are created. It is their output code which eventually becomes part of the parser/unparser subsystem.

Graphical View Code: Additional information needed for a graphical representation of a logical document is specified by the user of the meta environment in the form of annotations of the EBNF document (not shown in fig. 3). The annotations are used by the EBNF generator to produce a view on the logical document encapsulating the representation knowledge needed by the graphical unparser. Note that the overall look–and–feel of the generated environment is encapsulated by the general IPSEN Window System and UI Elements components, it is therefore identical for all environments.

The generated components described above, together with the invariant part of the framework, can be compiled and linked together to create a complete context–free graphical/textual editor for the respective language. The next step to be taken is to specify the *context–sensitive properties* of the language using the PROGRES language (part B in fig. 2).

As was shown in fig. 2, the EBNF again serves as input to a generator which provides a PROGRES frame specification containing the schema declaration and basic AST operations for the specified language, cf. fig. 3. This frame specification can be regarded as the PROGRES counterpart of the EBNF generator output described under logical document code. The meta environment user may then specify context–sensitive properties by extending the schema part and defining corresponding graph operations.

The resulting PROGRES specification is eventually *transformed into source code*. The major part of this code will be integrated into the logical document subsystem, resulting in an interface which respects the context–free as well as the context–sensitive properties of the supported language.

The process described so far yields an environment featuring a syntax–based editor for textual and/or graphical languages with free input facilities and an analyzer for the syntax and static semantics of the language. The next step the environment builder usually takes is to augment the set of language–unspecific tools provided by the framework (undo–redo tool, browser etc.) with *language–specific tools*, e.g. complex analyzing, monitoring, or execution tools. Again, it is possible to specify the tool operations with PROGRES, generate the corresponding source code, and to integrate the resulting tool into the framework architecture. Detailed information on this topic and examples, again, can be found in [16].

7. Summary

At the time being, the IPSEN meta SDE has a main focus on generating *single document processing* language–sensitive tools. Its meta tools for generating text–oriented or diagrammatic editors, unparsers, parsers, etc. were heavily used in the last years for the development of various software engineering environment prototypes as well as the meta environment itself. Examples include an extensive MIL environment [4], a requirements engineering environment featuring integrated data flow and EER diagrams [12], and an administrative environment for the management of products, processes, and resources [22].

Rather recently added extensions for generating complex context–sensitive editing operations, incrementally working type checkers, and execution tools are still under development. They rely heavily on the existence of the PROGRES environment. An EBNF translator takes a context–free grammar as input and produces an equivalent *graph rewriting system specification* as output. PROGRES tools may then be used to deal with context–sensitive tool operations and to translate their specifications into plain Modula–2 or C code.

Generating source code from the specification instead of tables to be evaluated at run–time restrains our approach in that a compile–link cycle is involved in any change to the specification. On the other hand, it is more flexible because tables can hardly express all the aspects of the specification. To yield both the flexibility of source code generation as well as the possibility to apply changes at run–time, a very complex and interpreted input language would be needed. In fact, a corresponding *interpreter* for the *PROGRES language* has been built, but its performance restricts its use to the validation of specifications rather than being usable in a "real" SDE tool.

All generated (Modula–2 or C) source code documents have their specific places in the overall *IPSEN framework architecture*. This framework offers all the functionality needed for building standard user interfaces and a hand-coded graph library with basic routines for implementing incrementally working analysis, interpreter, and compiler tools.

Finally, we should mention one major point of criticism concerning the current state of the presented meta SDE. No support is offered for generating application–specific parts of *integration tools*. Until now, some basic components have been realized which are very helpful for implementing tools of this category based on coupled graph grammars [13]. But the automatic translation of these *coupled graph grammar specifications* into the implementation of incrementally working transformation or better integration tools is the main subject of ongoing research activities.

References

[1] R. Bahlke, G. Snelting: "The PSG System – From Language Definitions to Interactive Programming Environments", in *TOPLAS 8, 4*, ACM Press, 1986, pp. 547–576

[2] R. Ballance, S.L. Graham, M.L. Van der Vanter: "The Pan Language–Based Editing System", in *TOSEM 1, 1*, ACM Press, 1992, pp. 95–127

[3] P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, V. Pascual: "Centaur: The System", in P. Henderson (Ed.): Proceedings of the 3rd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, *ACM Software Engineering Notes 13, 5*, ACM Press, 1988, pp. 14–24

[4] J. Börstler: "IPSEN: An Integrated Environment to Support Development for and with Reuse", in W. Schäfer et al. (Eds.): *Software Reusability*, Ellis Horwood, 1994, pp. 134–140

[5] V. Donzeau–Gouge, G. Huet, G. Kahn, B. Lang: Programming Environments Based on Structured Editors: "The Mentor Experience", in D. R. Barstow, H. E. Shrobe, S. Sandewall (Eds.): *Interactive Programming Environments*, McGraw Hill, 1984, pp. 128–140

[6] G. Engels, C. Lewerentz, M. Nagl, W. Schäfer, A. Schürr: "Experiences in Building Integrating Tools, Part I: Tool Specification", *TOSEM 1, 2*, ACM Press, 1992, pp. 135–167

[7] E. Golin, T. Magliery: "A Compiler Generator for Visual Languages", in *Proceedings Visual Languages '93*, IEEE Computer Society Press, 1993, pp. 314–321

[8] S. B. Horwitz, T. Teitelbaum: "Generating Editing Environments Based on Relations and Attributes", in *TOPLAS 8, 4*, ACM Press, 1986, pp. 577–608

[9] U. Kastens, M.W. Waite: "Modularity and Reusability in Attribute Grammars", in *Acta Informatica 31*, 1994, pp. 601–627

[10] N. Kiesel, A. Schürr, B. Westfechtel: "GRAS – A Graph–oriented (Software) Engineering Database System", in *Information Systems 20, 1*, 1995, pp. 21–51

[11] P. Klint: "A Meta–Environment for Generating Programming Environments", in *TOSEM 2, 2*, ACM Press, 1993, pp. 176–201

[12] C. Kohring, M. Lefering, M. Nagl: "A Requirements Engineering Environment within a Tightly Integrated SDE", in *Requirements Engineering 1*, Springer–Verlag, 1996, pp. 137–156

[13] M. Lefering: "Software Document Integration Using Graph Grammar Specifications", in Proceedings of the 6th International Conference on Computing and Information, *Journal of Computing and Information 1, 1*, 1994, pp. 1222–1243

[14] M. Minas, G. Viehstaedt: "DiaGen: A Generator for Diagram Editors Providing Direct Manipulation and Execution of Diagrams", in *Proceedings Visual Languages '93*, IEEE Computer Society Press, 1995, pp. 203–210

[15] M. Nagl: *Softwaretechnik: Methodisches Programmieren im Großen*, Springer–Verlag, 1990

[16] M. Nagl (Ed.): *Building Tightly–Integrated (Software) Development Environments: The IPSEN Approach*, LNCS 1170, Springer–Verlag, 1996

[17] J. Paakki: "Attribute Grammar Paradigms – A High Level Methodology in Language Implementation", *ACM Computing Surveys 27, 2*, ACM Press, 1995, pp. 196–256

[18] T. Reps, T. Teitelbaum: *The Synthesizer Generator Reference Manual*, Springer–Verlag, 1988

[19] J. Rekers, A. Schürr: "Defining and Parsing Visual Languages with Layered Graph Grammars", to appear in: *Journal of Visual Languages and Computing, 8, 1*, Academic Press, 1997, 29 pages

[20] A. Schürr, A.J. Winter, A. Zündorf: "Graph Grammar Engineering with PROGRES", in W. Schäfer/P. Botella (Eds.): *Proceedings of the 5th ESEC*, LNCS 989, Springer–Verlag, 1995, pp. 219–234

[21] G. Snelting, R. Bahlke: "PSG: A Theory–Based Environment Generator", in N. H. Madhavji, W. Schäfer, H. Weber (Eds.): *SD&F1 – Proceedings of the 1st International Conference on System Development Environments & Factories*, Pitman, 1990, pp. 131–140

[22] B. Westfechtel: "Integrated Product and Process Management for Engineering Design Applications", in *Integrated Computer–Aided Engineering 3,1*, John Wiley & Sons, 1996, pp. 20–35