Graph Grammar Engineering: A Method Used for the Development of an Integrated Programming Support Environment

G. Engels, W. Schäfer
Angewandte Informatik, FB 6, Universität Osnabrück
Postfach 4469, D-4500 Osnabrück

Abstract

We introduce a method to specify the functional behaviour of software tools in an incremental and integrated software development environment. This specification method is based on graph grammars. It is an adequate method to specify the behaviour of all software systems using graphs as internal data structures. We show that a specification can be developed systematically by which the adaptability of the environment is increased towards modification of tools or extension by new tools. Furthermore, guidelines for the implementation can directly be derived from this specification.

Key Words: integrated tools, graph grammars, programming support environments, programming-in-the-small, software engineering, specification

1. Introduction

The systematic development of large software systems requires a precise description of its desired behaviour. Depending on the used method such a **specification** of the behaviour is a more or less formal description on a conceptual level (cf. /SF 76/). In this paper we introduce a specification method based on graph grammars. This method is shown to be adequate if graphs are the underlying data structures on the conceptual level. This means that the behaviour of the software system can be described by graph transformations.

The method is applied to the specification of the tools of an **I**ncremental **P**rogramming **S**upport **En**vironment (**IPSEN**) (cf. /Na 84/). The task of such an environment is to facilitate the development and maintenance of software documents. Software documents are for example (a piece of) a description of the modularization of a software system, the source code of a module, a technical documentation, etc.

Furthermore, we show that such an operational specification on a conceptual level also leads to a specification in a second sense denoting the result of the design phase in a software life cycle model. This specification determines the **decomposition** of the software system **into modules** and serves as the guideline for the implementation phase. So, the reader may be aware that we talk about two levels of the term 'specification'.

IPSEN in particular has some external characteristics which influence the design of the user interface.

(1) Input and output of software documents is always **syntax-directed** and **incremental**, i.e. it is done in logical portions of an underlying syntax definition of a class of software documents. Incrementality means that syntax-analysis, evaluation, or execution is even possible for partial programs, specifications, etc.
(2) The tools in IPSEN are **integrated** in different senses: (a) Most of the technical activities of the software life cycle are supported, (b) the tools are combining activities which nowadays are regarded to belong to different activities in life cycle models, (c) all tools have a uniform user interface, i.e. the user is not aware of an internal change between different tools.

This incremental and integrated mode implies that the sequential organisation of activities in usual life cycle models can no longer be sustained. For example, whenever an increment of the specification is put in (design phase), checks for the intermodular connections may immediately start (integration phase). Furthermore, this part of the specification may now be changed (usually in maintenance phase). Therefore, we have grouped the activities in the following task areas: **programming in the large** (any activities directed to the level above single modules), **programming in the small** (any activities directed to the level of single modules), and, finally, **project organisation and project management.**

The incremental and integrated mode yields the following main characteristic of IPSEN. All information contained in an external representation of a module, specification, etc. is represented in and can be accessed from a high-level intermediate data structure. As the conceptual model for this data structure we use graphs. The reason is that graphs are a uniform model which can be applied to any task area. In particular, we have a **system graph** as intermediate representation of a module decomposition, a **module graph** for a single module, a **documentation graph** for technical documentation, etc. Structural information is expressed by labeled nodes and labeled edges whereas only nonstructural information is expressed by additional attributes. Aspects of editing, evaluating and execution of software documents can be treated by this model as further information for any purpose can be expressed by graphs without leaving the class of graphs. The uniform model heavily facilitates the development of integrated tools. Any kind of modification of these graphs by any tool can be specified by graph grammars.

By listing the internal characteristic we have also mentioned the main difference between IPSEN and other programming environments. These environments rather independently developed similar integrated concepts (cf. Gandalf /Ha 82/, Mentor /DG 80/, Cornell Program Synthesizer /TR 81/, Pecan /RE 84/). However, using graphs and not trees as intermediate data structures and graph grammars as specification instrument is specific to IPSEN. In all other approaches trees are regarded as the main conceptual model for the intermediate data structure which yields a lot of problems. A big part of the structural information has to be expressed in attributes as it cannot be expressed in trees. This yields partially complex attribute evaluation algorithms (cf. /RT 83/). Furthermore, additional and quite different complex internal data structures are introduced, when extending the environment to new tools (e.g. Pecan /Re 84/).

Please note that this is not an argument for avoiding attributes at all. Attributes are necessary for expressing values. We pledge for using the same graph model for all structural information.

In this paper we mainly deal with the specification of the **integrated tool set** for **programming in the small** by **graph grammars.** Such a tool set (cf. /Na 84/) consists of e.g. a syntax-aided editor, a tool for static analysis of the data flow in a module, and a tool for static instrumentation by e.g. the insertion of breakpoints. A module is executed by interpreting the corresponding module graph. This execution is supported by e.g. tools for runtime data inspection resp. for establishing specific environments for the test of procedures.

Since all tools use the module graph as internal data structure, such a graph has to represent a lot of different tool-specific information. Thus, it yields a complex problem to specify the behaviour of all tools by one graph grammar. We show that this **complexity** can be **decreased** by developing an independent graph grammar specification for each tool in a first step, and by combining these graph grammars in a second step. Thereby, this also serves as a **guideline** for the **efficient** implementation of the different tools. Such a **specification engineering** using graph grammars is the crucial point of this paper. It was applied to all tools for the task area programming in the small. So, the composition of these specifications is our main concern here, whereas the systematic development of a single graph grammar specification, namely the syntax-aided editor, was described in /EG 83/.
The proceeding presented here can be analogously transferred to other tools and task areas in IPSEN. For a specification editor this is documented in /LN 84/. The specification engineering is one result of the IPSEN project: It is more the development of new concepts for the specification and implementation of a programming support environment which is our main goal than an industrial implementation.

The **organisation** of the **paper** is as follows: The next section describes the systematic development of the module graph based on the syntax of the programming language Modula-2. (Modula-2 is chosen as the programming language to be supported as well as implementation language, because it offers adequate concepts for programming in the small and programming in the large.) In section 3 the modification of this module graph by different tools is specified using graph grammars. Section 4 gives an idea, how this specification has to be changed to get more efficient module graph modifications. In section 5 it is shown how this formal specification of the functional behaviour directly leads to different implementation techniques of the tools. Section 6 summarizes the main ideas of this paper.

2. The Module Graph

As mentioned in the introduction the module graph serves as the common, high-level data structure for all tools of the area programming in the small. It is an abstract representation which contains all tool-specific informations. In a module graph all structural information is expressed by different labeled nodes and edges, while all

non-structural information is expressed by additional tool-specific attributes.

Because of the incremental mode in IPSEN the module graph itself is a composition of **graph increments.** Since module graphs have to represent modules written in Modula-2, the graph increments heavily depend on the underlying programming language grammar. Such a grammar is usually designed with respect to certain properties, e.g. an efficient deterministic context free syntax analysis. This causes the grammar to contain a lot of technical nonterminals, which do not reflect logical portions of the language. In order to get reasonable graph increments these technical nonterminals have to be eliminated. We introduce a normal form for such a string grammar respecting the logical portions of the syntax corresponding to the increments.

2.1 Normalized Backus-Naur Form (BNF)

Such a string grammar **normal form** is characterized by a disjoint decomposition of the set of nonterminal symbols into three groups. (We use the BNF-notation to describe the productions of a context free string grammar.)
The first group contains the so-called **alternative nonterminals** and **optional nonterminals.** The right-hand side of all productions with an alternative nonterminal as left-hand side consists of a series of alternatives all of which are nonterminal or terminal symbols (cf. fig. 2.1). The right-hand side of the productions corresponding to optional nonterminals consists of two alternatives, the empty word and a nonterminal describing the optional part.
The second group is built by so-called **compound nonterminals** describing a part of the 'structure' of a module. The right-hand side of the corresponding productions is one sequence of nonterminal and terminal symbols (cf. fig. 2.1). The third group contains **list nonterminals** describing non-empty lists of elements of a certain kind (e.g. statement_list).

```
alternative nonterminal:
<statement> ::= <assignment_statement> | <while_statement> | <for_statement> | ...

structure nonterminal:
<while_statement> ::= while <expression> do <opt_statement_list> end
```

Fig. 2.1: Nonterminals and productions of a normalized Modula-2 BNF

It is the task of the IPSEN designer to transform a given programming language grammar into such a normal form. Since the resulting grammar is not uniquely determined, the grammar designer indirectly also determines the form of a module graph. It is to be seen in the next section, how this grammar influences the module graph increments.

2.2 Graph Increment Classification

Graph increments are associated with each compound or list nonterminal and its corresponding production, respectively. These **compound** and **list graph increments** consist of a root node (labeled by an indication for the corresponding increment) and as many sons, as nonterminals on the right-hand side of the production or list elements exist. In

compound graph increments the edges between the root node and its sons are labeled additionally according to their semantics (cf. fig. 2.2). In list graph increments the order of the list elements is expressed by additional edges labeled by 'ord' (cf. fig. 2.3).

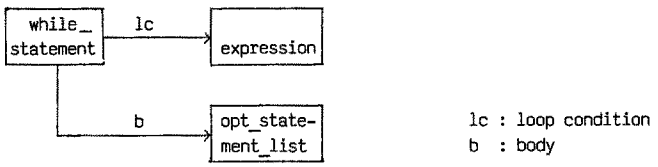

Fig. 2.2: compound graph increment

Since the lexical structure of identifiers or literals is not interesting for any IPSEN tool it will not be represented in a module graph. Therefore, identifiers and literals are represented by so-called **simple graph increments.** These increments consist only of a single node attributed with a string representing a concrete identifier or literal (cf. fig. 2.3).

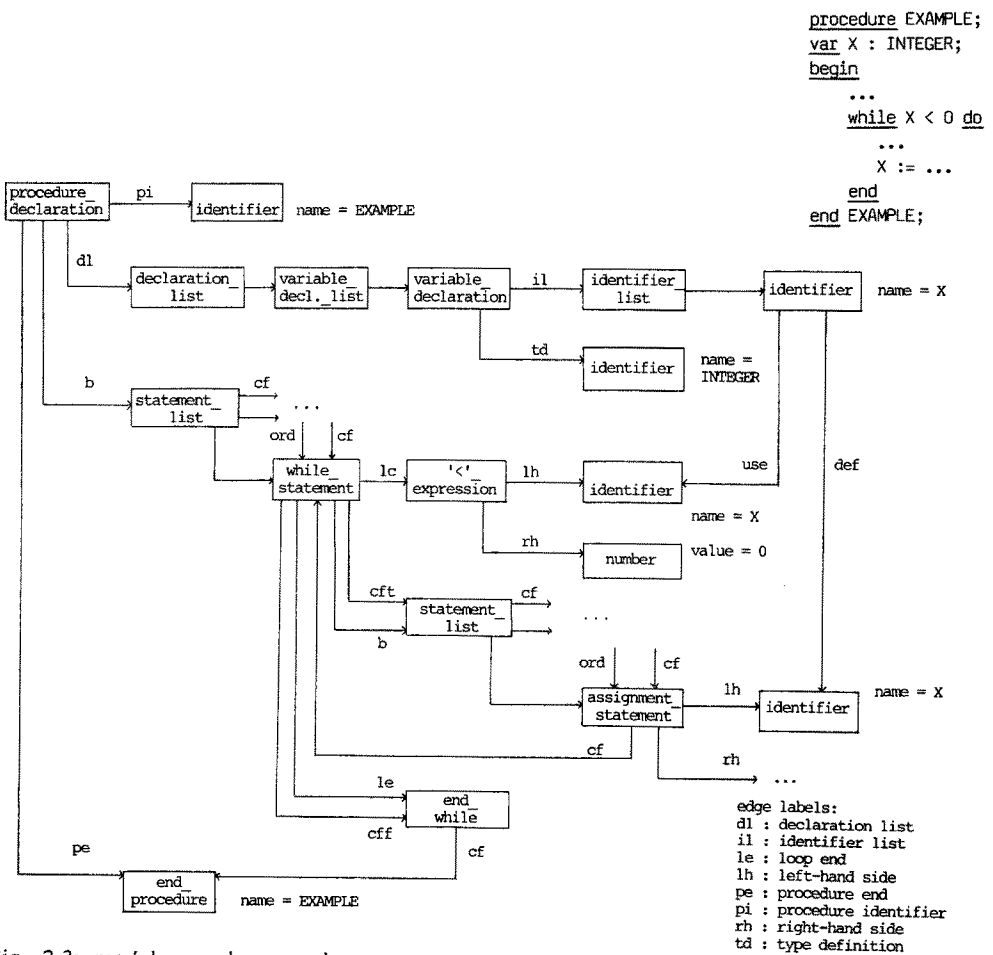

Fig. 2.3: module graph example

Neglecting the ord-edges between list elements the composition of this graph increments yields a tree which is well-known as the abstract syntax tree in literature (cf. e.g. /DG 80/). It forms the spanning tree of the module graph and reflects the structural information of a module given by the context free syntax. Concrete syntax is not represented in the graph as it expresses no structural information. It will be the task of an unparser to generate source text including the concrete syntax from this abstract representation. Figure 2.3 presents the module graph representation of an incomplete Modula-2 procedure.

2.3. Module Graph Supplements

Besides the relations expressed by the spanning tree further relations between the graph increments are expressed by additional edges. For example, to ease and speed up **context sensitive** checks after an incremental modification of a module we introduce edges between the declaration of a variable and its defining (the variable is getting a new value) and using occurrencies (the value is read) in the statement part. The defining occurrencies are connected by a 'def'-edge with the declaration, whereas the using occurrencies are connected by a 'use'-edge with the declaration (cf. fig. 2.3).
By the composition of graph increments all allowed control flows in a module are determined. This information is expressed in a module graph by three different labeled edges, the cf-edge for unconditional control flow, and the cft- (control flow true) or cff- (control flow false) edge if the control flow depends on a condition (cf. fig. 2.3).

Of course, these additional edges in a module graph can be read by all tools working on the module graph. Besides that, all tools may introduce further edges and nodes to express their **tool-specific** structural informations. Examples are edges to express the data flow in a module (set-use edges), or the textual order of a corresponding source text (unparsing edges).

In the other projects mentioned in the introduction the internal data structure is a tree so that all these relations have to be expressed by additional attributes. The lack of those approaches is that the inherent structure of a software document is described in different notions, namely trees and attributes. In IPSEN only one formal notion, namely graphs, is used to denote structure. Only non-structural information is expressed by attributes. Examples are values of literals, unparsing information as concrete syntax, storage addresses of data objects, etc..

3. Specification of Programming Support Tools

The commands of the area programming in the small allow a user to edit a module, to analyze it, to execute it, to instrument it by different test facilities, etc. These commands are part of an integrated tool set, i.e. the user is not aware of an internal change e.g. between the editor and the interpreter.
The execution of such a user command internally implies the activation of one tool action or for more comfortable user commands a sequence of tool actions. Since all tools work on the same internal data structure, the module graph, each tool action

implies a sequence of module graph modifications. These modifications are done incrementally. This means that after a modification of a graph increment in the module graph by one tool, all other tool-specific informations of this graph increment are immediately updated, too. So, the term 'incremental' occurs in two senses: the incremental behaviour of all tools at the user interface on one side and the incremental updating of the module graph on the other side.

In this section we show that an operational specification of such an integrated and incrementally working tool set can be developed by graph grammars in several steps. At first, the modification of graph increments and the incremental updating of other tool-specific informations has to be specified. Afterwards each tool can be specified independently by combining these modifications of the common internal data structure. At the end, the specification of all tools can be combined in one common graph grammar. This graph grammar is the operational specification of the integrated tool set.

Such a systematic proceeding reduces the complexity of specifying a large integrated tool set and, furthermore, increases the adaptability towards the modification of a tool or the extension by new tools.

Each execution of a tool action may be considered as a module graph transformation together with a corresponding evaluation of tool-specific attributes. Therefore, **attributed graph grammars** are a formal method to specify the behaviour of tools. Such attributed graph grammars consist of a set of attributed graph rewriting rules, called productions, containing an embedding rule and attribute evaluation instructions. The application of productions is defined by replacing one occurrence of the left-hand side in the host graph by the right-hand side. The embedding rule defines how the replacing graph has to be connected to the host graph. Attribute evaluation instructions are short Modula-2 code pieces, here usually simple assignments. They have to be executed as part of the application of an attributed graph grammar production. For a more formal introduction of graph grammars the reader is referred to /Na 79/ and /Bu 81/.

The position in a module actually handled by the user is marked by a special cursor node in the module graph. This implies that each production contains this cursor node on both sides, and that the occurrence of the left-hand side in the host graph is determined uniquely and can be found efficiently.

The combination of tools resp. the execution of one tool action often implies a determined order of different tool actions resp. of different graph modifications. Such a determined order of graph grammar production applications can be specified by so-called **control procedures** written like Modula-2 procedures the bodies of which contain the activation of other control procedures or productions. Therefore, we are using programmed sequential and attributed graph grammars.

Let us show now that these graph grammars are an adequate method for specifying the internal behaviour of programming support tools and their integration in a programming support environment.

The specification consists of three **layers**, a data structure oriented lowest layer and two functional layers upon that.

The **lowest layer** provides all control procedures and productions specifying all elementary modifications of the module graph. This includes the insertion and deletion of graph increments, and the modification of additional, tool-specific edges, nodes, and attributes. The control procedures and productions are gathered up in different graph grammars, for example Graph_Increment_Productions, Control_Structure_Supplements, Source_Text_Supplements. Examples of such productions are given in the following figures. The embedding rule will be omitted because it is the identity.

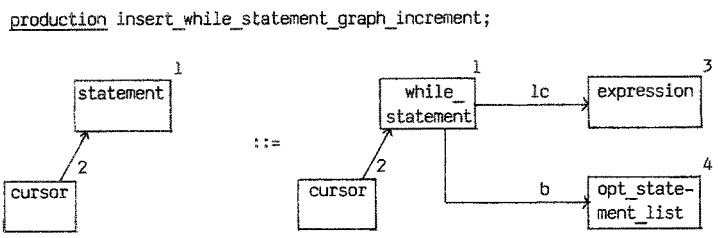

Fig. 3.1: production of graph grammar Graph_Increment_Productions

We add an 'end'-node to each graph increment representing a control structure, where the control flow of that control structure flows together:

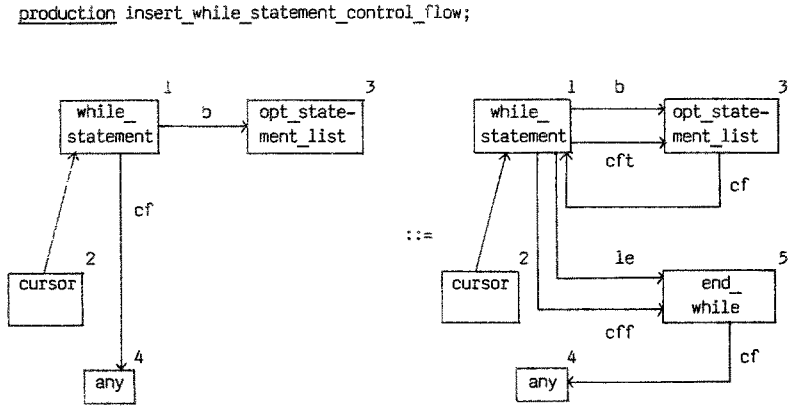

Fig. 3.2: production of graph grammar Control_Structure_Supplements

As an example for the graph grammar Source_Text_Supplements we consider the information needed by an unparser to generate source text from the module graph. Therefore, the graph increments have to be connected by further edges labeled by 'u' reflecting the textual order, and node attributes have to be added containing unparsing schemes describing the missing concrete syntax and the layout of the corresponding source text.

```
production insert_while_statement_unparsing_supplements;
```

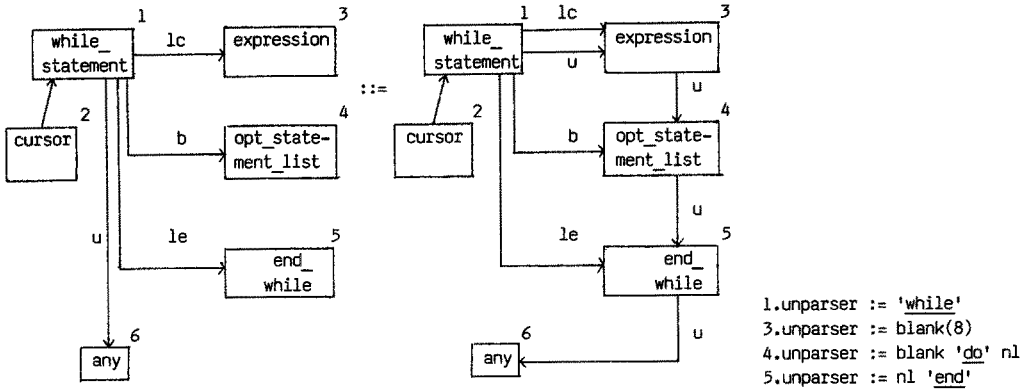

Fig. 3.3: production of the graph grammar Source_Text_Supplements

In the **second layer** all tools will be specified by independent graph grammars. These graph grammars consist of control procedures determining the order of applications of control procedures and productions of graph grammars of the lowest layer.

As an example we specify the insertion of a while-statement and the actualisations of other tool-specific informations by a control procedure of the tool Syntax_Aided_Editor.

```
procedure insert_while_statement_and_actualize;
begin
   insert_while_statement_graph_increment;
   insert_while_statement_control_flow_supplements;
   insert_while_statement_unparsing_supplements;
end insert_while_statement_and_actualize;
```

Fig. 3.4: control procedure of graph grammar Syntax_Aided_Editor

Such control procedures may also contain checks like identifier_allowed to test whether a specific graph is contained in the module graph. This enables testing the context sensitive syntax. For a detailed and systematic description of the development of the graph grammar Syntax_Aided_Editor we refer to /EG 83/.

The behaviour of all other tools can be specified analogously in separate graph grammars. Some tools like the Interpreter or Unparser additionally walk through the module graph. Therefore, such tools use productions of the graph grammar Cursor_Movements to move the cursor node in the module graph.

Since each tool was specified in a separate graph grammar a **third layer** is needed for specifying their combination. This means to specify functional dependencies which can be done by control procedures adequately. As an example we indicate the control procedure that specifies the execution of a while-statement using a loop-counter. In control procedures a simple user interface behaviour can be specified, too.

```
procedure execute_while_loop_with_loop_counter;
var number: integer;
begin
   insert_loop_counter; (* control procedure of static_instrumentation *)
   interprete_loop_with_loop_counter( number );
   user_message( 'number of loop executions:', number );
end execute_while_loop_with_loop_counter;
```

Fig. 3.5: control procedure of the graph grammar Tool_Handler

The whole functional composition / decomposition of graph grammars and the distinction between these three layers is illustrated in figure 3.6.

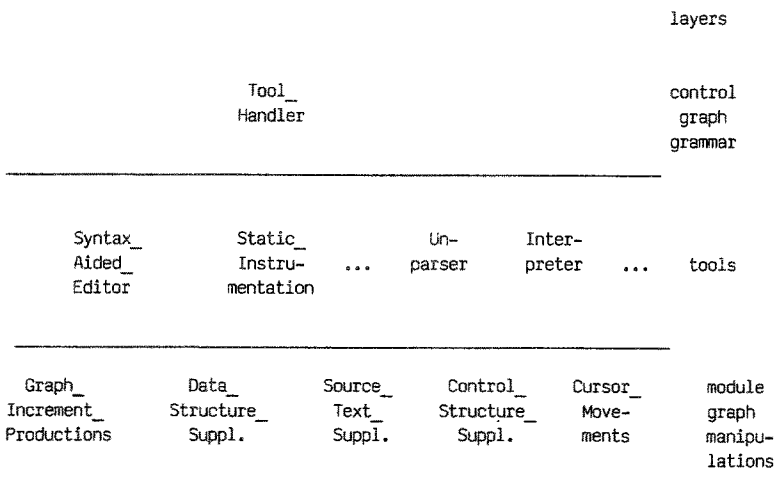

Fig. 3.6: the composition of the control graph grammar Tool_Handler

Such a layered approach of a graph grammar specification enforces a slightly modification of the usual definition of applicability of graph productions (cf. /Na 79/). That definition requires that a production or test is only applicable to a host graph iff the left-hand side is a **subgraph** of the host graph. But, to enable the sequential application of the productions of fig. 3.1 - 3.3 in the control procedure of fig. 3.4, we require that the left-hand side is contained as a **partial graph** in the host graph. Such a modification allows each tool a restricted, tool-specific view of the graph increments and the independent construction of the control procedures in the two functional layers.

Summarizing the approach we can say that the graph grammar Tool_Handler contains all control procedures and productions of the first and second layer and combines the control procedures of the second layer such that each command given by a user corresponds to a control procedure.
This formal composition / decomposition of graph grammars reflects the composition / decomposition of tools of an integrated tool set. Each command given by a user corresponds to a control procedure of this graph grammar Tool_Handler.

Besides this functional composition / decomposition we also have a data-structure oriented composition / decomposition, namely the composition / decomposition of a module graph of / into graph increments and tool-specific informations. The reader should be aware that this composition /decomposition is described by the graph grammar Tool_Handler, too.

4. Condensation of Graph Grammar Productions

The concept of an independent graph grammar specification for each tool, as explained in section 3, sometimes yields some inefficiences in the following sense.
Each control procedure of the graph grammar Tool_Handler was formed by combining the control procedures of the different tools. This combination was done by sequentially calling control procedures of the different tools one after the other (cf. fig. 3.5.). Because of this sequential calling mechanism it often happens that a lot of control procedures resp. productions and tests to be called in one combining control procedure change the underlying module graph in the same locality in several, consecutive steps. Same locality means that a certain graph increment (or a partial graph of this) is contained in the left-hand side of nearly all productions.

As the implementation is strongly related to the operational graph grammar specification (cf. section 5), the above mentioned situation causes some inefficiences which unneccessarily increase runtime, i.e. an implementation has to find a (partial) graph increment at any time a further tool specific control procedure resp. a further production is called. Obviously, it is much more efficient to search the common partial graph only once in the module graph, and then carry out all the modifications described by the according control procedure.

To realize this idea, we change our graph grammar specification. This modification is to 'summarize' as many productions as possible of one control procedure. It is done in two systematic steps which will be shown now.

In the **first step** we summarize a sequence of partial graph replacements into one partial graph replacement. This implies that the independent specification of each tool can no longer be sustained.

As an example we summarize the two productions 'insert_while_statement_control_flow' (cf. fig.3.2) and 'insert_while_statement_unparsing_supplements' (cf. fig.3.3) called by the control procedure 'insert_while_statement_and_actualize' (cf. fig.3.4).

The embedding rule informally says that all incoming edges of node 1 are identically transferred to node 1 on the right-hand side. All outgoing edges labeled with 'u' or 'cf' of node 1 become outgoing edges of node 5 of the right-hand side. The formalism for the notation of the embedding rules is borrowed from /NA 79/.

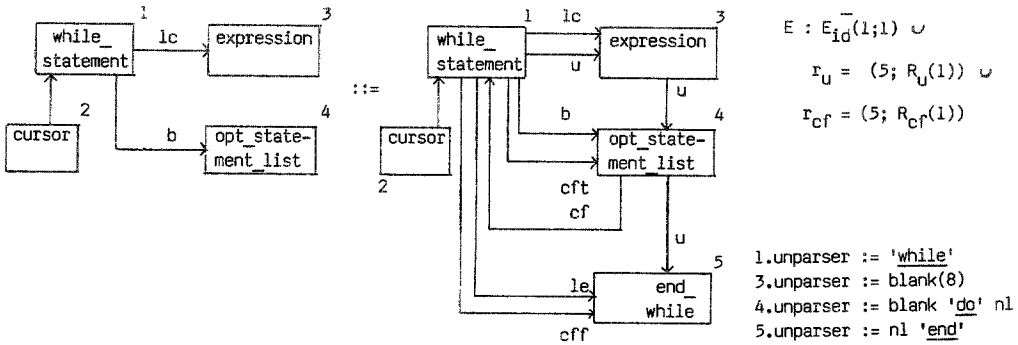

Fig. 4.1: example for the first step

The **second step** is based on the observation that the right-hand side of one production is identical to the left-hand side of other productions. In this case, the consecutive application of two productions can again be summarized. This situation often appears, when an increment is inserted in the module graph. As an example we regard the production 'insert_while_statement_graph_increment' (cf. fig. 3.1) and the above mentioned production (cf. fig. 4.1). By this step here, the right-hand side of the production in fig. 3.1 is replaced by the right-hand side of the production in fig. 4.1. The embedding rule and attribute transformation still hold.

After these two steps, the control procedure insert_while_statement_and_actualize now consists of only one production. The same steps can be done analogously for each such control procedure.

Unfortunately, the approach has two disadvantages. If you want to change the actions of one tool, or extend the programming environment by a new tool you will have to repeat these steps. (After a single graph grammar for the modified or new tool was developed.)
Furthermore, it has to be done by hand. The reason is that the condensation of productions heavily depends on their special shape, i.e. it is difficult to determine which edges are inserted by an embedding rule (cf. fig. 4.1), or which edges or nodes have to be added or omitted when mixing two productions (cf. fig. 4.1 the nodes marked with 'end-while' and 'expression').

Nevertheless, it is not useless to do these steps, as the advantage of this approach is that one can derive directly a more efficient implementation by using the same formalism as when specifying the different tools (cf. section 5).

5. Implementation Issues

The graph grammar Tool_Handler is an operational specification of all tools for programming in the small. In this chapter we show how such a specification directly leads to a guideline for the implementation of the tools. This guideline, i.e. a specification in the second sense, yields a further decomposition of the programming

environment. But now, this is a decomposition into modules to be used by the programmers writing the code of the implementation. We demonstrate that two different modularizations can be derived resulting in a more or less efficient implementation.

5.1 The Interpretative Approach

A first approach is to interprete the graph grammar by a universal graph grammar interpreter. However, the control procedures already have the shape of Modula-2 procedures. So, only the application of graph productions as well as the partial graph tests in boolean conditions of control structures must be handled by an interpreter, whereas the other part of the graph grammar can directly be translated by a Modula-2 compiler. (This, of course, was one of the reasons to choose the special notion of control procedures for programming in graph grammars.)

A rough overview on this part of the IPSEN-specification is given now in terms of a module concept developed in the IPSEN-project, too. Other parts, as e.g. the auxiliary components to realize the sophisticated and hardware independent I/O-handling, are described in /ES 84/ in more detail.

In this module concept we distinguish different types of modules. **Data type** modules encapsulate a data structure together with its operations, whereas **function** modules summarize a class of complex algorithms. These algorithms are based on operations of one or more data type modules as well as other function modules. We say, module A is usable in module B, if module B imports explicitly resources of the export interface of A. For further details of this concept we refer to /LN 85/.

The data structure 'graph' together with its operations is encapsulated in the data type module Graph. It exports resources like 'Insert node with label x and attribute y', 'Replace edge a by edge b', etc.. In reality this module is a rather big subsystem containing a graph storage, i.e. a system to store, retrieve and modify arbitrary graphs (cf. /BL 84/).

A second data type module GraGra_Productions provides a storage for arbitrary graph grammar productions and partial graph tests. In our case, all productions and tests of the lowest layer of the graph grammar Tool_Handler are stored.

A function module GraGra_Production_Interpreter exports resources to apply graph grammar productions and tests. To implement these resources this function module uses the two data types modules to read a certain production or test and to find the occurrence of a partial graph as well as to replace it.

The control procedures are implemented by a further function module Graph_Modification. Its implementation is given by the Modula-2 part of the control procedures. Interpretation of graph grammar productions and tests is done by use of resources of the module GraGra_Production_Interpreter.
According to the division in different graph grammars for each tool and a corresponding graph grammar in the highest layer this function module can be subdivided in different function modules for each tool and a coordinating function module upon them.

Analogously to the decreased complexity in the graph grammar specification this division yields a more elucid modularization.

Furthermore, all these function modules use a subsystem called User_Interface to realize the I/O-operations contained in the control procedures.

By this model of a graph grammar interpreter we get two main advantages: After having specified the different tools, one has the possibility of quickly testing these tools by using the interpreter (rapid prototyping). Furthermore, this specification can easily be changed by only changing one graph grammar. So, in this realization strategy the stress is layed upon adaptability and not on efficiency on a certain machine.

5.2 The Compilative Approach

Now we renounce the concept of interpreting the productions in order to get a more efficient implementation. Of course, it is a step towards inflexibility. So this step should only be done when the test phase is finished and the environment has to be tuned up.

What we do now is to **implement** any **application of a production** or **graph test** directly as a (Modula-2) program. Here, we do not search the left-hand side in the host graph by a partial graph test and then replace it by a right-hand side (both done by the interpreter). Instead, we directly 'implement' a graph rewriting step by inserting /deleting the nodes and edges which are the result of the application of a rule. Here, we also introduce the knowledge of the underlying class of graphs (in our case the module graph). Please note that the actual position of modification is internally indicated by the cursor node.

Such a procedure implementing the application of a special production or test uses the resources of the module Graph. Furthermore, the one to one correspondence between a production and a procedure need not to be sustained. Graph algorithms, namely special partial graph tests or replacements which are used for the application of many rules of the given graph grammar can be written as procedures and can be called in any application of the different rules.

Of course, this approach changes the design of IPSEN. The two modules GraGra_Productions and GraGra_Production_Interpreter are replaced by a function module. Its resources are procedure calls for the different productions of the given graph grammar which is again in our case the lowest layer of Tool_Handler. The application of different productions is implemented directly using the elementary graph operations of the module Graph.

6. Conclusions

We have indicated that graphs grammars are a well-suited specification method to describe the internal behaviour of an integrated set of programming support tools working on graphs as high-level data structures. As the resulting graph grammar is an operational specification which means programming on an 'abstract level', we can use software engineering methods like modularization and integration to decrease the

complexity of such an 'abstract' program. So, the specification of the tools can be done rather independently which makes the specification elucid and flexible both needed for modifying or adding tools. The complex problem of specifying a lot of tools on a quite complicated graph structure is decreased by a layered approach to the definition of such a graph grammar. The specification is also a guideline for the implementation of such an environment, i.e. it directly leads to a main part of the result of the design phase (also called specification).

The main topics of IPSEN are the development of such conceptual ideas as well as the implementation of a programming support environment on a minicomputer. Up to now, the graph storage (/BL 84/) and parts of the user interface (especially a window manager) are implemented. The graph grammar specification for the syntax-aided editor and most parts of the other tools for programming in the small (Interpreter, Static_Instrumentation, Unparser) are under elaboration and will be implemented soon in a prototype version of IPSEN.

Acknowledgements.

The authors are very indebted to M. Nagl and C. Lewerentz for many fruitful discussions.

References

/BL 84/ Brandes, Th./Lewerentz, C.: GRAS: A Non-standard Data Base System within a Software Development Environment, Tech. Rep. OSM - 118, Univ. of Osnabrueck

/Bu 81/ Bunke, H.: Attributed Programmed Graph Grammars as a Tool for Image Interpretation, Purdue University, Techn. Report TR-EE-81-22

/DG 80/ Donzeau-Gouge, M. et.al.: Programming Environments Based on Structured Editors - The MENTOR Experience, Techn. Report 26, INRIA, France

/EG 83/ Engels, G./Gall, R./Nagl, M./Schäfer, W. : Software Specification using Graph Grammars, Computing 31, 317-346

/ES 84/ Engels, G./ Schäfer, W.: The Design of an Adaptive and Portable Programming Support Environment, submitted for publication

/Ha 82/ Habermann, N. et.al.: The Second Compendium of GANDALF Documentation, Techn. Report, May 1982, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh

/LN 84/ Lewerentz, C./ Nagl, M.: A Formal Specification Language for Software Systems Defined by Graph Grammars, in U. Pape (Ed.): Proceedings WG'84 on 'Graphtheoretic Concepts in Computer Science', Linz: Trauner Verlag

/LN 85/ Lewerentz, C./Nagl, M.: Incremental Programming in the Large: Syntax-aided Specification Editing, Integration and Maintenance, to appear Proc. 18th Hawaii International Conference on System Sciences

/Na 79/ Nagl, M.: Graph-Grammatiken - Theorie, Anwendungen, Implementierung, Wiesbaden: Vieweg-Verlag

/Na 84/ Nagl, M.: An Incremental Programming Support Environment, to appear in Computer Physics Communications, North-Holland

/Re 84/ Reiss, St.: PECAN: Program Development Systems That Support Multiple Views, in Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh

/RT 83/ Reps, T./ Teitelbaum, T.: Incremental Context-Dependent Analysis for Language-Based Editors, ACM TOPLAS, Vol. 5, No. 3, 449-477

/SF 76/ Schnupp, P./ Floyd, Ch.: Software - Programmentwicklung und Projektorganisation, Berlin: Walter de Gruyter

/TR 81/ Teitelbaum, T./Reps, T.: The Cornell Programm Synthesizer - A syntax-directed Programming Environment, CACM 24 , 9 , 563-573