A Hybrid Interpreter in a Software Development Environment

Un interprétateur hybride dans un environnement de programmation

Gregor Engels
FB 17, Universität Mainz
Postfach 3980, D-6500 Mainz, West Germany

Andreas Schürr*
Informatik III, RWTH Aachen
Ahornstr. 55, D-5100 Aachen, West Germany

Abstract

This paper describes the realization of an execution tool for Modula-2 modules which is part of an integrated tool set in a software development environment termed IPSEN (Incremental Programming Support Environment). In this environment, all software documents, e.g. Modula-2 modules, are manipulated by syntax-directed editors and are represented internally by attributed graphs. The execution of Modula-2 modules is done by two cooperating interpreters. The first one is a graph interpreter which traverses the internal graph from statement to statement. These statements are translated into a low-level, more efficiently executable object code and interpreted by a second interpreter. This concept of a hybrid interpreter allows the realization of an execution tool which offers a lot of runtime support features to the user.

Résumé

Ce papier expose l'idée fondamentale d'interprétateur hybride, un outil pour exécuter des Modula-2 modules. Cet interprétateur fait partie d'un ensemble d'outils intégrés, qui forme l'environnement de programmation IPSEN (Incremental Programming Support Environment). Cet environnement se base sur le concept d'édition structurée guidée par la syntaxe du language, par exemple Modula-2, auquel appartiennent les documents manipulés. La représentation interne des tels documents est toujours un graphe décoré d'attributs. L'interprétateur hybride se compose de deux interprétateurs. L'un traverse en temps d'exécution le graphe représentant une module et traduit les instructions courantes au code objet bas de gamme, l'introduction pour l'autre interprétateur. Cette construction facilite l'execution efficace des Modula-2 modules et permet d'offrir bien des aides de mise au point aux utilisateurs de l'environnement de programmation IPSEN.

*The author is supported by Stiftung Volkswagenwerk

Key words: software development environment, programming-in-the-small, integrated tool set, testing support, interpreter, incremental compiler, Modula-2, attributed graph

Mots-clé: l'environnement de programmation, programmation en détail, l'ensemble d'outils intégrés, aides des mise au point, l'interprétateur, compilateur incrémental, Modula-2, graphe décoré d'attributs

CONTENTS

1 Introduction

The **IPSEN**-project (**I**ncremental **P**rogramming **S**upport **En**vironment) is aimed at the design and realization of an integrated software development environment (SDE) containing tools that support nearly all activities during the software life cycle ([Na 80], [Na 85]). This means that the environment offers tools to support programming-in-the-small with Modula-2, programming-in-the-large, documentation control as well as project management. In all these task areas syntax-directed editors, analyzers, instrumentation tools, and execution tools have been developed or are under development. This paper reports on new results concerning the realization of an execution tool for the programming-in-the-small task area. Other aspects of the IPSEN-project are described by several other publications ([ES 85a], [ENS 87], [LN 85]).

The main characteristic of SDEs like IPSEN is the **integration** of all tools at the user interface as well as in their internal realization. This means for the programming-in-the-small area that frequent switchings from editing to executing and testing activities have to be supported at the user interface, and have to be realized time-efficiently. Therefore, the traditional batch-oriented, compilative approach is not adequate for the realization of an execution tool in a SDE. A more suitable approach is a **conversational** or **incremental compiler**, first proposed by Lock ([Lo 65]) in the 60's.

The idea behind incremental compilers is to avoid the recompilation of the whole source text if the user has modified only a small portion. A wide spectrum of different concepts for the realization of such compilers can be found in the literature. All of them consist of two parts: the **syntax analysis**, producing a high intermediate data structure as an internal representation of the source text, and the **code generation**. Incremental syntax analysis is usually performed by a syntax-directed editor which modifies the internal representation and guarantees the syntactical correctness (cf. [DH 84]). Incremental code generation means that only a special portion of the object code is replaced after a source text modification. The size of this portion ranges from the level of whole modules or so-called compilation units ([DoD 80]) down to the level of procedures ([Ha 82], [SDB 84]) or even single statements ([EC 72], [Fr 84]).

However, before the execution can be started the whole source text has to be translated into object code if the compilative approach is used. This procedure eventually generates object code parts that may be discarded due to subsequent editing activities and that, therefore, are never needed (cf. [SDB 84]). Furthermore, the execution of incomplete source texts is not possible and the integration with other tools of a SDE is difficult to realize (e.g. [AMN 81]). Therefore, an **interpretive approach** is pursued in some SDE projects (e.g. [TR 81]). In this case, the incremental syntax analysis part is combined with an interpreter which directly interprets the internal representation of the source text. Thus the disadvantages of a compilative approach are avoided, but time-efficiency is impaired.

In order to make use of the advantages of an interpretive approach without loosing time-efficiency, we have realized the execution tool in IPSEN as a mixture, consisting of a statementwise incremental compiler and an interpreter. The main characteristics of our approach in comparison to others are:

- Internal representation of the source text by a graph-like intermediate data structure instead of abstract syntax trees and additional symbol tables. This data structure allows an incremental context-free and context-sensitive syntax analysis (cf. section 2).
- Realization of the execution tool by a hybrid interpreter combining two cooperating interpreters. The first one is a graph interpreter which directly interprets the graph-like internal representation and activates at specific points a second one - the object code interpreter - which interprets an incrementally generated object code (cf. section 3).
- Integration of the hybrid interpreter with other tools. This results in a sophisticated debugging environment for the programming-in-the-small task area (cf. section 4).

2 Incremental Syntax Analysis

In IPSEN, incremental syntax analysis is realized by a syntax-directed, command-driven editor ([ENS 87]), [Sc 86]). This editor uses a graph-like data structure, termed **module graph,** as an internal representation of a Modula-2 module. In order to avoid consistency problems, in IPSEN this module graph is used as the common data structure for all tools in the programming-in-the-small task area. This approach differs from others that use several data structures, e.g. abstract syntax tree, symbol table, calling graph ([Fr 84], [Rei 84], [Sc 72], [TR 81]).

The skeleton of a module graph is an **abstract syntax graph** (similar to an abstract syntax tree (cf. [DH 84])) representing the context-free structure of a Modula-2 module.

This abstract syntax graph is enriched by further edges expressing context-sensitive relations between syntactical increments in a Modula-2 module. These edges are required by the syntax-directed editor in order to report and prevent immediately not only context-free but also context-sensitive errors. All these additional edges represent the context-sensitive information that is usually stored in the so-called **symbol table.** There are, for instance, edges from the applied occurrences of an object identifier to its declaration (cf. Figure 1).

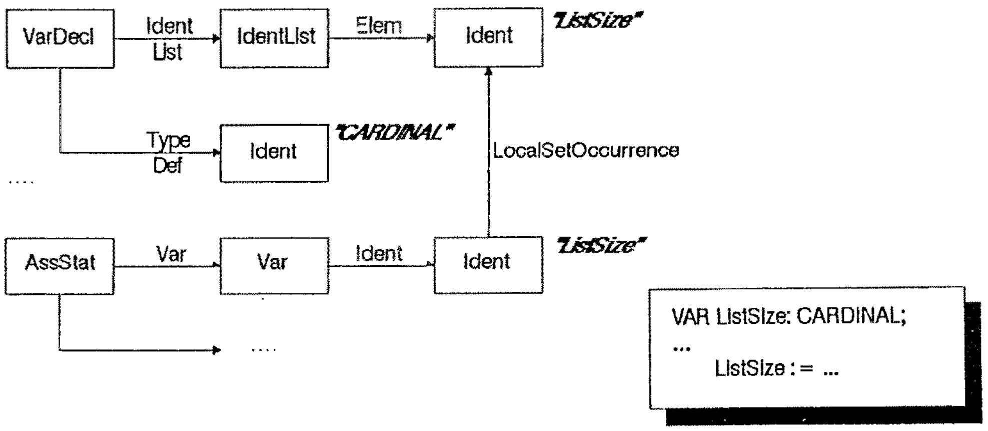

Figure 1: cutout of a module graph

Furthermore, tool-specific information is also stored in the module graph. For instance, additional edges are used to express the control flow in statement parts (cf. Figure 2).

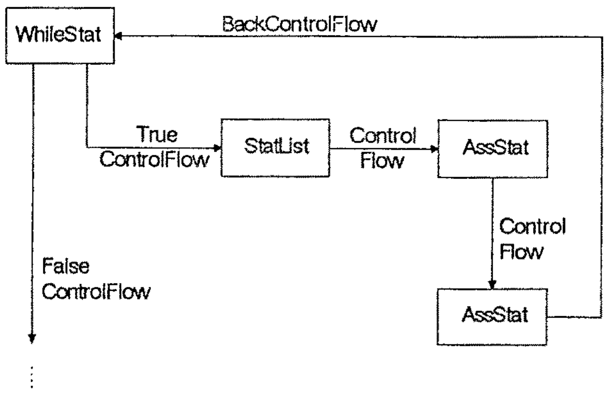

Figure 2: control flow edges in a while-statement

All the nodes and edges in the module graph expressing context-free and context-sensitive relations or other tool-specific information are **immediately updated** after a modification of the source text is made. This guarantees the context-free as well as context-sensitive correctness of the module graph at any time.

3 Incremental Code Generation

To execute Modula-2 modules, internally represented by module graphs, we have developed a hybrid interpreter([Sa 86]). This interpreter combines two cooperating interpreters with a code generating component.

The dominant part of the hybrid editor is a **module graph interpreter** which traverses the module graph along the control flow edges. This approach is similar to that used in PECAN

([Rei 84]). Whenever this interpreter enters a procedure for the first time, a **declaration evaluator** is activated. This component determines the mapping of the local data objects to the runtime storage. And whenever this module graph interpreter reaches an execution increment for the first time, a **code generator** produces the corresponding object code. Execution increments are, for instance, assignment-statements or conditional expressions of while-statements. The generated code, in turn, is interpreted by a second interpreter, called the **object code interpreter** (cf. Figure 3).

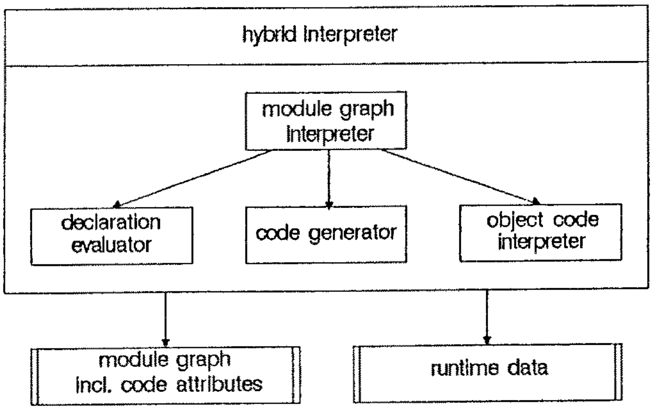

Figure 3: overall structure of the hybrid interpreter

Declaration evaluation and code generation can be done easily, since the context-sensitive edges in the module graph directly represent the binding of identifiers to their declarations. As a consequence, we do not have to build up a separate symbol table at runtime to generate object code or to interprete this code ([AMN 81], [Sc 72], [TR 81]).

The output of the declaration evaluator and the code generator is stored in specific node attributes of the module graph. The address attribute of variable identifiers or the code attribute of execution increments are good examples.

We use common compiler construction techniques, well-known and efficient, to allocate storage for runtime data on a stack and a heap. This runtime data storage is not embedded in the module graph, but realized by a separate data structure.

The object code interpreter is a usual abstract stack machine with type-dependent instructions for the standard Modula-2 types and arithmetic operations. This interpreter is a variant of the P-Code interpreter ([PD 82]) enhanced by special control transfer instructions. By these instructions the control from the object code interpreter is returned to the module graph interpreter, whenever the interpretation has to proceed to a new execution increment in the graph. Generating abstract object code instead of directly producing machine code has the advantage that the implementation of the hybrid interpreter is independent of the target machine. On the other hand, generating machine code, directly or by an additional compilation step, could also be provided (cf. [Ro 83], [SDB 84]). This would result in a more time-efficient interpretation.

In section 2 we have explained that all context-sensitive edges are incrementally updated after each editor action. Such an incremental approach could also be applied to all execution specific attributes. But this would cause the permanent (re-)computation of attribute values which might never be needed. This is only tolerable in a multi-tasking, sinle-user operating system (cf. [SDB 84]). Therefore, we decided to compute and store these attribute values just at the moment when the hybrid interpreter actually requires their values for the first time. This procedure raises two questions for the module graph interpreter:

- Is a required attribute value already stored?
- Is a stored attribute value invalid due to some editing of the module graph?

To solve these problems we introduced auxiliary boolean attributes in order to distinguish valid execution specific attributes from invalid ones. Determination of the invalid attributes is done by starting at the changed parts of the graph and following context-free as well as context-sensitive edges through the module graph. Invalid attributes are (re-)computed, when the execution needs them for the next time. This strategy is similar to a two-phase attribute evaluation strategy ([Re 84]) with a delayed second phase. Long traversals up and down an abstract syntax tree are avoided by using additional context-sensitive edges to establish new paths through the module graph.

4 Debugging with the Hybrid Interpreter

To make possible **integrated** testing and editing of Modula-2 modules the hybrid interpreter has to cooperate with the following other software tools in the IPSEN environment. These are the **syntax-directed editor**, a **static analyzer** for informing the user about a lot of context-sensitive relations in a module (e.g. set/use chains), and a **testing preparation tool** for instrumenting a module (e.g. loop counter, conditional breakpoints). 'Integrated' means that the user can easily switch between these different tools in a modeless way and activate any command of any tool at any time.

The use of incremental compilation techniques renders possible the efficient cooperation of the hybrid interpreter with the syntax-directed editor and the testing preparation tool.

The mixture of two interpreters makes possible the realization of a user friendly debugging interface with the following characteristics:

- The granularity of execution steps (statementwise, blockwise) can be changed interactively.
- The current position of the execution is marked in the displayed source text.
- The current execution can be interrupted at any point by pressing a special key.
- Unfinished source texts can be executed. If a gap in the source text (e.g. a missing expression in a while-statement) is encountered during the execution, the user is allowed to fill the gap with the syntax-directed editor. Afterwards, the execution proceeds automatically.
- Error messages (e.g. use of undefined data objects), single variable values or larger portions of the runtime data can be displayed in terms of the source text.

- All additional instrumentation of the source text forms a testing environment that can incrementally be modified, and switched on and off at runtime.

5 Conclusions

To sum up, editing and executing Modula-2 modules in IPSEN are realized by four interlacing phases:

1. Incremental editing of an attributed module graph, representing a context-free and context-sensitive correct Modula-2 module.
2. Traversing the module graph along the control flow edges.
3. Lazy evaluation of node attribute values representing the storage mapping function and the object code.
4. Interpretation of the generated object code.

This concept enables us to implement a conversational and time-efficient execution tool as an integral part of a SDE. Up to now, a prototype implementation of all tools of the programming-in-the-small task area has been realized. All tools are implemented in Modula-2, and they run on an IBM AT 02 under the operating system MS-DOS. The whole implementation has a total of 50.000 lines of code. The realization of the hybrid interpreter consists of 5.000 lines of code. Up to now, only a subset of Modula-2 is supported by the tools of the programming-in-the-small area. Extending this subset and the hybrid interpreter to an execution tool of the programming-in-the-large task area is one aspect of the current work in the IPSEN-project.

Acknowledgements
The authors are indebted to all members of the IPSEN team: Chr. Beer, Th. Janning, C. Lewerentz, M. Nagl, A. Sandbrink, W. Schäfer, U. Schleef, and B. Westfechtel.

References

[AMN 81] Atkinson, L.V./ McGregor, J.J./ North, S.D.: Context sensitive editing as an approach to incremental compilation, The Computer Journal, Vol. 24, No. 3, 222-229

[DH 84] Donzeau-Gouge, V./Huet, G./Kahn, G./Lang, B.: Programming Environments Based on Structured Editors: The Mentor Experience, in: Barstow, D.R. et al. (eds.): Interactive Programming Environments, McGraw-Hill

[DoD 80] DoD: Reference Manual for the Ada Programming Language, LNCS 106, Berlin: Springer

[EC 72] Earley, P./Caizergues, A.: A Method for Incrementally Compiling Languages with Nested Statement Structure, CACM, Vol. 15, No. 12

[En 86] Engels, G.: Graphs as Central Data Structures in a Software Development Environment, Ph.D. thesis, (in german), Düsseldorf: VDI-Verlag

[ENS 87] Engels, G./ Nagl, M./ Schäfer, W.: On the Structure of Structure-Oriented Editors for Different Applications, in Proc. of the Second Symposium on Practical Software Environments, Palo Alto, ACM SIGPLAN Notices, Vol.22, No. 1, 190-198

[ES 85a] Engels, G./Schäfer, W.: Graph Grammar Engineering: A Method Used for the Development of an Integrated Programming Support Environment, in LNCS 186, 179-193, Berlin: Springer

[ES 85b] Engels, G./Schäfer, W.: The Design of an Adaptive and Portable Programming Support Environment, in Valle, G./ Bucci, G. (eds.): Proc. of the International Computing Symposium 1985, Florence, Italy, 297-308, Amsterdam: North-Holland

[Fr 84] Fritzson, P.: Preliminary Experience from the DICE System, A Distributed Incremental Compiling Environment, in ACM SIGPLAN Notices, Vol. 19, No. 5, 113-123

[Ha 82] Habermann, N. et al.: The Second Compendium of Gandalf Documentation, technical report, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh

[LN 85] Lewerentz, C./Nagl, M.: Incremental Programming in the Large: Syntax-aided Specification Editing, Integration and Maintenance, in Proc. of the 18th Hawaii International Conference on System Sciences, Vol. 2, 638-649

[Lo 65] Lock, K.: Structuring Programs for Multiprogram Time-Sharing On-Line Applications, in: Proc. AFIPS, FJCC 27, 457-472

[Na 80] Nagl, M.: An Incremental Compiler as Part of a System for Software Development, in IFB, Vol. 25, 29-44, Berlin: Springer

[Na 85] Nagl, M.: An Incremental Programming Support Environment, in Computer Physics Communications 38, 245-276, Amsterdam: North-Holland

[PD 82] Pemberton, St./ Daniels, M.: Pascal Implementation: The P4 Compiler, Chichester: Ellis Horwood Ltd.

[Rei 84] Reiss, St.P.: An Approach to Incremental Compilation, Proc. of the ACM SIGPLAN '84 Symp. on Compiler Construction, SIGPLAN Notices, Vol. 19, No. 6, 144-151

[Re 84] Reps, Th.: Generating Language Based Environments, MIT-Press

[Ro 83] Robson, D.J.: An evaluation of throw-away compiling, Software - Practice and Experience, Vol. 13, No. 3, 241-149

[Sa 86] Sandbrink, A.: Design and Implementation of a Testing and Runtime Support Regarding Interpreter, Master's thesis, (in german), University of Osnabrueck

[Sc 86] Schäfer, W.: An Integrated Software Development Environment: Concepts, Design, and Implementation, Ph.D. thesis, (in german), Düsseldorf: VDI-Verlag

[Sc 72] Schmidt, H.A.: A User Oriented and Efficient Incremental Compiler, in Proc. Int. Comp. Symp., Venice, 259-269

[SDB 84] Schwartz, M.D./ Delisle, N.M./ Begwani, V.S.: Incremental Compilation in Magpie, Proc. of the ACM SIGPLAN '84 Symp. on Compiler Construction, SIGPLAN Notices, Vol. 19, No. 6, 122-131

[TR 81] Teitelbaum, T./Reps, Th.: The Cornell Program Synthesizer: A Syntax-Directed Programming Environment, in: CACM, Vol. 24, No. 9, 563-573