Parallel programming structures and attribute grammars*

R. ALVAREZ GIL and Á. MAKAY

Kalmár Laboratory of Cybernetics, Áprád tér 2, H-6720 Szeged, Hungary

1. Introduction

The attribute grammars are useful tools to give the semantics of programming languages for compiler construction, thus many complier generators based on attribute grammars have been developed [4] [7] [8] [9] [11] [12].

Many papers deal with attribute grammars describing structure of sequential languages for compiler construction, but only a few deals with parallel programming structures.

In this paper we give the semantics of the bracket pair **cobegin-coend** and the symbol **and** in words, and afterwards we give the object which the parallel programming constructions will be translated to. In section 3 we give an attribute grammar able to perform the required translation. The concept of attribute grammars and the notations used can be seen in [1]. In section 4 we mention some experiences got in the implementation by means of attribute grammars of a parallel programming language in which processes communicate through Hoare's monitors.

The methods given in the paper were tested successfully in a CDC 3300 computer of the Hungarian Academy of Sciences with the help of the HLP/SZ compiler generator system [11].

* Supported by the Research Foundation of Hungary, Grant No. 1066, 1143.

2. Semantics and translation of the bracket pair cobegin-coend and the constructor and

The constructor **and** is used to separate instructions such as the symbol;, but the instructions separated by **and** may be executed in parallel. The priority of the symbol **and** is higher than the priority of the symbol;. Thus in the following part of a program: statement_1; statement_2 **and** statement_3; statement_4, statement_2 and statement_3 are executed parallel, but after finishing the execution of statement_1 and before beginning the execution of statement_4.

The bracket pair **cobegin-coend** is used to ecnlose a statement list such as the bracket pair **begin-end**, but the statement of a statement list enclosed in a bracket pair **cobegin-coend** are executed parallel. To separate the statements enclosed in **cobegin-coend**'s can be used; as well as **and** or mixing the two symbols.

For the translation of these parallel programming constructions we will use three primitives: **fork**, **join** and **quit** [2] [3]. We have selected these primitives because the operations **fork** and **quit** are available in all languages including the psosibility to creating and terminating processes, while **join** can be realized by a "go to" statement and a semafor.

Execution of the operation **fork** w creates a new process starting at the statement labelled w. If a process executes a primitive **join** t, w it is equivalent with $t:=t-1$; **if** $t=\emptyset$ **then goto** w as a unique and indivisable operation.

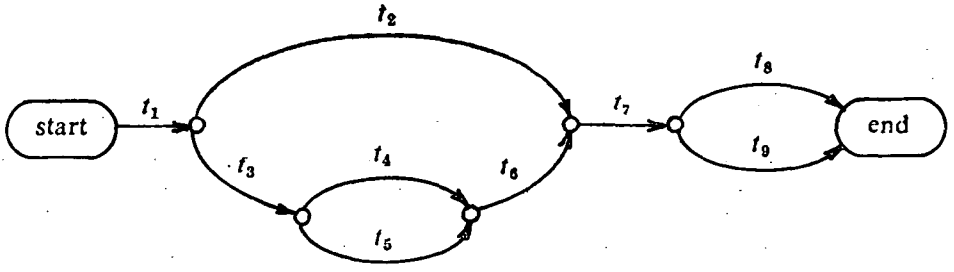
To determine the tasks statically we have to decompose the program into segments representing processes or parts of processes. Of course, processes are not uniquily determined. For example let's see the following program:

begin statement_1; statement_2; statement_3;
cobegin begin statement_4; statement_5 **end**;
begin statement_6; statement_7 **and** statement_8;
statement_9; statement_{10} **end**
coend;
statement_{11}; statement_{12};
begin statement_{13}; statement_{14} **end and** statement_{15}
end

This program can be partitioned into the following segments:

t_1: statement_1; statement_2; statement_3
t_2: statement_4; statement_5
t_3: statement_6
t_4: statement_7
t_5: statement_8
t_6: statement_9; statement_{10}
t_7: statement_{11}; statement_{12}
t_8: statement_{13}; statement_{14}
t_9: statement_{15}.

Moreover we can associate to the program a task flow graph [10] in which each edge corresponds to the execution of a segment:



One of the possibilities to translate our program partitioned into those segments with the above primitives is the following:

begin t_1: statement$_1$; statement$_2$; statement$_3$; n_1:=2; **fork** t_2; **quit**;
t_2: **fork** t_3; statement$_4$; statement$_5$; **join** n_1, t_7; **quit**;
t_3: statement$_6$; n_3:=2; **fork** t_4; **quit**;
t_4: **fork** t_5; statement$_7$; **join** n_3, t_6; **quit**;
t_5: statement$_8$; **join** n_3, t_6; **quit**;
t_6: statement$_9$; statement$_{10}$; **join** n_1, t_7; **quit**;
t_7: statement$_{11}$; statement$_{12}$; n_7:=2; **fork** t_8; **quit**;
t_8: **fork** t_9; statement$_{13}$; statement$_{14}$; **join** n_7, end; **quit**;
t_9: statement$_{15}$; **join** n_7, end; **quit**;
end: **end**

An attribute grammar is able to define that kind of decomposition into segments and of translation to processes.

3. An attribute grammar to describe parallel programming structures for compiler construction

The translation of a structure **cobegin** statement$_1$; ...; statement$_n$ **coend** or statement$_1$ **and** ... **and** statement$_n$ will be as follows:

free m; t:=n; **fork** s_1; **quit**;
s_1: **fork** s_2; **occ** m_1, m_1'; ... code of statement$_1$...; **free** m_1;
join t, end; **quit**;
⋮
s_n: **nop**; **occ** m_n, m_n'; ... code of statement$_n$ ...; **free** m_n;
join t, end; **quit**;
end: **occ** m, m';

where **free** an **occ** are newly introduced macros to allocate and deallocate work-areas for processes.

We use the well known attributes "codelength" (synthesized) and "codeloc" (inherited) which give the length and the localization of the generated code. Another synthesized attribute is "level" to calculate the size of the work-area necessary for each process.

The code generation of parallel structures can be performed at the root of the subtree associated with them in the derivation tree after the generation of the code of

each segment (statement$_1$, ..., statement$_n$). For the generation of the correct primitives and macros it is enough to know the size of the work-area and the localization of each statement, because the localization of a statement can be used to obtain the label of the work-area of the statement.

We use some other attributes in the grammars. The synthesized attribute "csloc" gives the necessary information (the localization of the generated code and the size of the work-area of each statement) upwards to the root of the subtree. The inherited attribute "loclev" is a pair (m, m') giving the label and the size of the work-area which has to be allocated at the beginnig and has to be deallocated at the end of the execution of a parallel structure. The inherited attribute "costat" tells us whether a statement is in a parallel structure or not.

The code generation can be performed by a synthesized attribute which is to be evaluated during the last pass. We do not deal with it, because it would have a long and trivial description in the 4-th, 7-th and 8-th syntactical rules of the attribute grammar. Furthermore in a syntactical rule $p: X_0::=X_1 \dots X_{n_p}$ we will omit the semantical rules of the form $X_0.a=X_j.a$ $(1 \leqq j \leqq n_p)$ when there is no other X_i $(1 \leqq i \leqq n_p$ and $i \neq j)$ which has the same attribute "a", and also the rules of the form $X_j.a=X_0.a$ $(1 \leqq j \leqq n_p)$.

Now see the attribute grammar:

Nonterminal symbols and their attributes:
program has no attributes
block has codelength, level, codeloc, loclev
coblock has codelength, codeloc, loclev
stat_list has codelength, level, csloc, codeloc, loclev, soctat
statement has codelength, level, codeloc, loclev, costat
partstat_list has codelength, csloc, codeloc

Syntactical rules with their semantical rules:

i) program::=block
block.codeloc=2
block.loclev=(1, block.level)

ii) program::=coblock
coblock.codeloc=1
coblock.loclev=(0, 0)

iii) block::=**begin** stat_list **end**
stat_list.costat=false

iv) coblock::=**cobegin** stat_list **coend**
coblock.codelength=stat_list.codelength+5
stat_list.codeloc=coblock.codeloc+4
stat_list.loclev=(0, 0)
stat_list.costat=true

v) stat_list$_1$::=statement; stat_list$_2$
stat_list$_1$.codelength=statement.codelength+stat_list$_2$.codelength
stat_list$_1$.level=$\begin{cases} \text{statement.level, if statement. level} \geqq \text{stat\_list}_2\text{.level} \\ \text{stat\_list}_2\text{.level, if statement. level} < \text{stat\_list}_2\text{.level} \end{cases}$
stat_list$_1$.csloc=((statement.codeloc, statement. level), (a_1, b_1), ..., (a_k, b_k)),
where $((a_1, b_1, \dots, (a_k, b_k))$=stat_list$_2$.csloc
stat_list$_2$.codeloc=stat_list$_1$.codeloc+statement.codelength

vi) stat_list::=statement

stat_list.csloc=((statement.codeloc, statement.level))

vii) $stat\_list_1$::=parstat_list; $stat\_list_2$

$$stat\_list_1.codelength = \begin{cases} parstat\_list.codelength + stat\_list_2.codelength + 5, \\ \quad \text{if } stat\_list_1.costat = false \\ parstat\_list.codelength + stat\_list_2.codelength, \\ \quad \text{if } stat\_list_1.costat = true \end{cases}$$

$stat\_list_1.csloc = ((a_1, b_1), \ldots, (a_k, b_k), (c_1, d_1), \ldots, (c_l, d_l))$, where
$((a_1, b_1), \ldots, (a_k, b_k)) = parstat\_list.csloc$ and
$((c_1, d_1), \ldots, (c_l, d_l)) = stat\_list_2.csloc$

$$parstat\_list.codeloc = \begin{cases} stat\_list_1.codeloc + 4, \text{ if} \\ \quad stat\_list_1.costat = false \\ stat\_list_1.codeloc, \text{ if} \\ \quad stat\_list_1.costat = true \end{cases}$$

$$stat\_list_2.codeloc = \begin{cases} stat\_list_1.codeloc + \\ parstat\_list.codelength + 5, \text{ if} \\ \quad stat\_list_1.costat = false \\ stat\_list_1.codeloc + \\ parstat\_list.codelength, \text{ if} \\ \quad stat\_list_1.costat = true \end{cases}$$

Note: in this syntactical rule there is code generation if $stat\_list_1.costat = false$

viii) stat_list::=parstat_list

$$stat\_list.codelength = \begin{cases} parstat\_list.codelength + 5, \\ \quad \text{if } stat\_list.costat = false \\ parstat\_list.codelength, \\ \quad \text{if } stat\_list.costat = true \end{cases}$$

stat_list.level=0

$$parstat\_list.codeloc = \begin{cases} stat\_list.codeloc + 4, \\ \quad \text{if } stat\_list.costat = false \\ stat\_list.codeloc, \\ \quad \text{if } stat\_list.costat = true \end{cases}$$

Note: in this syntactical rule there is code generation if stat_list.costat=false

ix) $parstat\_list_1$::=statement **and** $parstat\_list_2$

$parstat\_list_1.codelength = statement.codelength + parstat\_list_2.codelength$

$parstat\_list_1.csloc = ((statement.codeloc, statement.level), (a_1, b_1), \ldots, (a_k, b_k))$, where $((a_1, b_1), \ldots, (a_k, b_k)) = parstat\_list_2.csloc$

statement.loclev=(0, 0)

statement.costat=true

$parstat\_list_2.codeloc = parstat\_list_1.codeloc + statement.codelength$

x) parstat_list::=statement$_1$ **and** statement$_2$

parstat_list.codelength = statement$_1$.codelength + statement$_2$.codelength

parstat_list.csloc = ((statement$_1$.codeloc, statement$_1$.level), (statement$_2$.codeloc, statement$_2$.level))

statement$_1$.loclev = (0, 0)

statement$_1$.costat = true

statement$_2$.codeloc = parstat_list.codeloc + statement$_1$.codelength

statement$_2$.loclev = (0, 0)

statement$_2$.costat = ture

xi) statement::=block

$$\text{statement.codelength} = \begin{cases} \text{block.codelength} + 5, & \text{if statement.costat} = \text{true} \\ \text{block.codelength}, & \text{if statement.costat} = \text{false} \end{cases}$$

$$\text{block.codeloc} = \begin{cases} \text{statement.codeloc} + 3, & \text{if statement.costat} = \text{true} \\ \text{statement.codeloc}, & \text{if statement.costat} = \text{false} \end{cases}$$

$$\text{block. loclev} = \begin{cases} (\text{statement.codeloc} + 1, \text{statement.level}), & \text{if statement.costat} = \text{true} \\ \text{statement.loclev}, & \text{if statement.costat} = \text{false} \end{cases}$$

xii) statement::=coblock

$$\text{statement.codelength} = \begin{cases} \text{coblock.codelength} + 5, & \text{if statement.costat} = \text{true} \\ \text{coblock.codelength}, & \text{if statement.costat} = \text{false} \end{cases}$$

statement.level = 0

$$\text{coblock.codeloc} = \begin{cases} \text{statement.codeloc} + 3, & \text{if} \\ \text{statement.costat} = \text{true} & \\ \text{statement.codeloc}, & \text{if} \\ \text{statement.costat} = \text{false} & \end{cases}$$

The method given here was tested in the CDC 3300 computer of the Hungarian Academy of Sciences with the help of the HLP-SZ compiler generator system. A sequential programming language was augmented with the bracket pair **cobegin-coend** and the symbol **and**, and we have produced a compiler based on an ASE (alternating semantics evaluator) attribute evaluation strategy [6] which has the same number of passes (five) as the compiler generated for the basic sequential language has. This fact and the introduction of only three new attributes show us that the complexity of a compiler based on an ASE strategy does not increase by the introduction of the parallel structures discussed here.

4. Some remarks about the implementation of processes communicating through Hoare's monitors

We have implemented a very simple experimental language in which parallel processes communicate through Hoare's monitors [5]. The language is block structured, and the scope rule for monitors is the usual: a monitor reference can appear in the block where the monitor was declared, or in a block contained in it. The structure of a block is the following:

begin
declarations of monitors local to the block;
declarations of variables local to the block;
... the block body ...
end;

A declaration of monitors has the form:

monitor $m_1, m_2, \ldots, m_n$ **of** m;

and creates the monitors $m_1, m_2, \ldots, m_n$ of type m, where m is a monitor type declared at the beginning of the program. For simplicity each monitor type must be declared at the beginning of the program. (In other implementations monitor types could be declared at the beginning of the blocks with the same scope rule of monitors). The structure of a monitor type is the following:

type monitor_type_name **monitor**;
begin
declaration of the condition variables;
declarations of variables local to the monitor;
procedure procedure_name (...formal parameters...);
declarations of the normal parameters;
begin
...the procedure body...
end;
...declarations of other procedures local to the monitor...;
...initialization of local data of the monitor...
end;

In the implementation of the experimental language each monitor has its local data area which contains the variables of the monitor, the queues of processes waiting on a condition or on a monitor call, and the queue of processes waiting after an issue of a signal operation.

We have to introduce many new attributes. Four of them are the most important, and they will be described here: the synthesized attributes MTL and MINTRN, and the inherited attributes LMT and MTOTAL.

The attribute MTL is used to construct a table in which informations are collected about the declared monitor types. We put into the table the following informations about each monitor type:

— monitor type name;
— list of the variables local to the monitor type;
— list of the condition variables of the monitor type;
— list of the procedures local to the monitor type which contains on each procedure the parameters of the procedure, the name of the procedure, and the list of condition variables which appear in a "wait" statement in the procedure;

— the object code of the initialization of the data local to the monitor and the length of the code.

The attribute LMT leads the table (the address of the table) from the root dawnwards the leafs of the derivation tree.

The attribute MINTERN is used to construct a table collecting information about the declared monitors. We put into the table the following informations about each monitor:

— the monitor name;
— the monitor type of the monitor;
— the number of condition variables of the monitor and the number of variables local to the monitor;
— addresses and lengths of the queues of the condition variables of the monitor;
— address and length of the queue of processes waiting in a monitor call;
— address and length of the queue of processes waiting by an executed "signal" statement.

The attribute MTOTAL gives the table of the monitors valid in the environment with respect to the scope rule for monitors.

The HLP/SZ is based on the programming language SIMULA, so we can use classes and objects, and attributes of type reference to work with tables. In other compiler generator systems based on attribute grammars the concept of global attribute is introduced to make it easy to work with tables.

Abstract

This paper gives an attribute grammar for the translation of parallel programming structures: the bracket pair **cobegin-coend** and the symbol **and**. The introduction of these constructions into a programming language does not increase the complexity of a compiler based on an ASE attribute evaluation strategy. We discuss the implementation of Hoare's monitors by means of attribute grammars. The methods given here were tested in a CDC 3300 computer of the Hungarian Academy of Sciences.

References

[1] Alvarez, R.: Giving mathematical semantics of nondeterministical and parallel programming structures by means of attribute grammars. Acta Cybernetica, Tom. 7. Fasc. 4. 1986. 413—423.
[2] Conway, M.: A multiprocessor system design. Proc. AFIPS 1963, Fall Joint Comput. Conf., 24. Spartan Books, New York, 139—146.
[3] Dennis, J. B. and von Horn, E. C.: Programming semantics for multiprogrammed computations. CACM 9, 3 (March 1966), 143—155.
[4] Ganzinger, H., Ripken, K. and Wilhelm, R.: Automating Generation of Optimizing Multipass Compilers. In Information Processing '77, North-Holland Publ. Co., 1977, 535—540.
[5] Hoare, C. A. R.: Monitors: An Operating System Structuring Concept. CACM 17, 10 (October 1974), 549—557.
[6] Jazayeri, M. and Walter, K. G.: Alternating Semantic Evaluator. In Proc. of ACM 1975 Ann. Conf., 230—234.
[7] Kastens, U.: GAG: A Practical Compiler Generator. Lecture Notes in Computer Sciences 141, 1982.
[8] Lewi, J., De Vlaminck, K., Huens, J. and Huybrechts, M.: A Programming Methodology in Compiler Construction. Part 1: Concepts. North-Holland Publ. Co., 1982.

[9] RAIHA, K. J., SOARINEN, M., SOISOLON-SOINEN, E. and TEINARI, M.: The Compiler Writing System HLP (Helsinki Language Processor), Department of Computer Science, Report A-1978-2, University of Helsinki.
[10] SHAW, A. C.: The logical design of operating systems. Prentice Hall Inc., 1974.
[11] SIMON, E. and GYIMÓTHY, T.: Attributum nyelvtanok és alkalmazásuk. Akadémiai Pályamunka. MTA Automataelméleti Tanszéki Kutató Csoport, Szeged, 1983.
[12] MAKAY, Á., GYIMÓTHY, T., SIMON, E.: An implementation of the HLP. Acta Cybernetica, Tom. 6. Fasc. 3, 1984. 315—327.

(Received January 11, 1989)