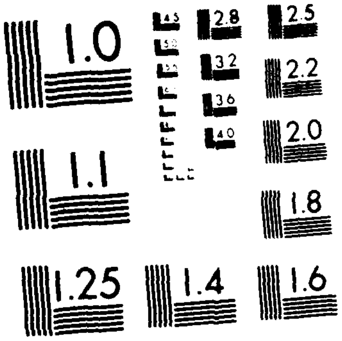
AD-A191 066   BEYOND ADA - GENERATING ADA CODE FROM EQUATIONAL   1/1
SPECIFICATIONS(U) RENSSELAER POLYTECHNIC INST TROY NY
DEPT OF COMPUTER SCIENCE B K SZYMANSKI 1987
UNCLASSIFIED N00014-86-K-0442   F/G 12/5   NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Beyond Ada - Generating Ada Code from Equational Specifications

by
Boleslaw K. Szymanski

Computer Science Department
Rensselaer Polytechnic Institute
Troy, NY 12128

N00014-86-K-0442

ABSTRACT

Real time mission-oriented embedded systems are much more difficult to design than ordinary software systems. They require highly reliable and efficient implementations to satisfy mission and time constraints imposed by the applications. The Ada language has been design to facilitate real time system software development. However, for many programmers the size and complexity of Ada itself are of concern.

In the assertive programming paradigm, computations are specified as sets of assertions about properties of the solution, and not as a sequence of procedural steps. Solving procedures are automatically generated from the assertive description. Real time programming for mission-oriented systems is supported by equational languages in which assertions are expressed as algebraic equations. Programs written in equational languages are concise, free from implementation details, and easily amenable to verification and parallel processing. The level of programming expertise required to program in an equational language is much lower than the level that is needed by Ada programmers.

The paper describes an implementation of an equational language system which generates highly efficient distributed code in Ada. It also demonstrates how the equational language system can be used in real time software development.

1. INTRODUCTION

Real time system programming is distinct from programming other parallel or distributed applications in that timing constraints are imposed on delays caused by real time programs. The complexity and diversity of skills needed for real time programming have caused extended development times, difficulties in attaining desired reliability and sometimes even a reluctance to undertake maintenance and updating of real time systems. This has motivated development of several programming languages [Brinch 1978, 1981, Martin 1978, Wirth 1977, and most notably Ada, 1978] to make the task easier.

Real time system development can often be simplified if it is done on higher programming level then supported by Ada. Several specification languages [Lamport 1983, Laner 1979, Lee 1986, Milner 1980, Ramamrithan, Teichreow 1977, Zave 1982] have been proposed to this end. Some of these languages support assertive programming paradigm which provide a bridge between the formal requirements of a real time system and the system implementation (for example in Ada). In this paradigm a computation is expressed as a set of assertions about properties of the solution and not as sequence of procedural steps.

In the paper, we discuss the design of an Ada code generator for the assertive language called MODEL [Tseng et al, 1986]. Assertions in MODEL are expressed as recursive equations. MODEL specifications are concise, free from implementation details, and easily amenable to verification and parallel processing.

The MODEL language and system aid or automate the following steps of the software development and maintenance process:

1. Generating high level language code for individual program units. A very high level, nonprocedural language (MODEL) is provided for writing the software specifications. The MODEL compiler uses specifications to generate program code in Ada or other high level programming language (Fortran, C or PL/1).
2. Establishing synchronization and communication between program units executing in parallel. The Configuration Specification Language (CSL) is provided for this purpose. A MODEL subsystem called Configurator generates communication tasks with necessary entries.
3. Testing. An executable model of the system that runs on the host computer is produced by the MODEL compiler and Configurator. This model can be used for testing, debugging and performance study purposes.
4. Documenting. Several reports are generated automatically. The following is a partial list: the system design and structure, individual program listing, generated Ada (or Fortran, C, or PL/1) code listing and timing reports.
5. Static timing performance analysis. Normally, the timing study can be done only after programs in target machine code have been produced and executed. Instead, with the help of a MODEL subsystem called timing evaluator, performance analysis can be done when an individual task has been specified, even on a host other than the target machine.

The paper discusses an implementation of the Ada code generator for the MODEL system. It is organized as follows. In the next section, we describe real time software development using MODEL. Section 3 discusses implementation of the Configurator and generation of communication tasks. Section 4 describes Ada code generation for program units. Finally, the last section offers the conclusion regarding the use of equational languages for real time programming.

2. REAL TIME SYSTEM DEVELOPMENT USING MODEL

In the MODEL approach, the programmer initially partitions the problem into units based on functional affinity. Then, each unit function is described in the MODEL equational language. A series of translators is employed to implement the computation and provide the feedback on performance. This guides the programmer in further partitioning or consolidating parallel units until a satisfactory, locally optimal, performance is reached.

Three software tools were developed to support our approach:

i) A compiler for the configuration specification language in which units' interconnections and a mapping of the parallel tasks onto processors are defined.

ii) A compiler for the MODEL equational language in which individual units are defined. This compiler produces parallel tasks for the respective processors.

iii) A timing evaluator for estimating the delays inherent in the parallel tasks. The estimates are used by the programmer to verify that the time constraints of the developed system are satisfied.

The real time software development process starts after the software system requirement are available. These requirements usually consist of three parts :

1. Functional requirements - defining the functions and subfunctions of the system.
2. Performance requirements - time constraints for time-critical performance of the system.
3. Definition of interfaces with the environment - the layout of the data communicated with the environment.

The programmer begins by dividing the system functions into software units and data files. A function may be carried out by one one or more units, or several related functions may be combined into one unit. The relationship and communications between units are also defined at this point. The program units are in skeletal form, with only the external data structures outlined as files. The programmer can now use the Configurator to verify global system consistency and completeness.

Next, the programmer composes the unit specifications independently for each program unit in the MODEL language. The MODEL compiler processes each unit separately, performing completeness and consistency checks within each unit, and in the absence of errors, generates an Ada program to perform the task of that unit. The user can now employ the Timing Evaluator on each generated Ada program to verify whether the time constraints associated with the corresponding program unit are satisfied. The Timing Evaluator produces a Timing Report for each program unit that provides information on time delays between instances of input and/or output in the unit. The user has to provide certain timing data of the target machine to the Timing Evaluator for it to generate the Timing Report.

The programmer may also have to check if global time constraints are met by adding individual unit delays in a path of the configuration to obtain the overall delays between critical events involving multiple units. If some of these constraints are not satisfied, the programmer may have to modify the configuration of the entire system by partitioning some units to obtain a greater degree of parallelism.

Once all the program units have been satisfactorily processed by the MODEL compiler and the timing evaluator, the programmer uses the Configurator to synthesize all the system components (units and data files) into an integrated system. The user composes the system by specifying a configuration of units and files in the Configuration Specification Language that is input to the Configurator. It then schedules individual program units, synchronizes units that will execute in parallel, generates tasks responsible for exchanging communications, and generates a configuration procedure that will run the Ada programs with maximum concurrency in the host computer's multiprogramming / multiprocessing environment.

Finally, the system can be executed and tested on the host machine. Then, code can be transferred to the target machine for further testing and execution.

3. CONFIGURATION SPECIFICATION LANGUAGE

The Configuration Specification Language, CSL, defines flow of data between program units. Objects of the language are units and files that the units exchange [Shi et al, 1987]. A target/source or consumer/producer relationship between a file (file) and a unit is represented by a directed edge between those objects. When the same file is produced by one unit and consumed by another, then these two units become connected via the file.

Two attributes of configuration nodes are worth of mentioning here. A *unit type* shows whether the unit is:

1) simple - an individually specified unit (default),
2) compound - a group of units for which a configuration is defined separately, or
3) interactive - a human communicating with the system through a terminal.

per letter

A-1

Files have an *organization* attribute with the following values: sequential (default), indexed, mail and post.

A sequential file is exchanged as one entity. It can be consumed only after it has been entirely produced. Such a file may have only one producer, but any number of consumers.

An indexed file has a variable defined as a key used to define (access) records in the file. There are no restrictions on the order or number of references to such a file made by producers and consumers.

A mail file is a collector of records. It is private to its consumer and therefore it can have only one consumer, but several producers. Records from different producers are accepted by the consumer in order of their arrival.

A post file is a distributor of records to dynamically addressable files. The post file has one producer, and its record include a key used as an address of a destination file. Therefore, it can have any number of edges connecting it to mail files.

An exchange of data between units executed in parallel can be set either through a mail file or a pair of a post and mail files connected together. The goal of our approach is to eliminate timing considerations from real time programming. The user's view of computation is totally static, where computation itself is expressed as a mapping of source data structures onto target data structures. Consequently, our communication primitives are based on (limited) nonblocking 'send' and blocking 'receive'. The producer of the messages continues computation immediately after of messages waits until the message to be read arrives. Such semantics allows the user to treat communications in exactly the same way as other i/o. If the synchronization is needed, it can easily be achieved by adding a 'receive' (in the producer unit) after the 'send' to obtain an answer (or an acknowledgement) from the consumer.

The MODEL compiler, when generating a program for a unit, optimizes the use of the main memory assigned to data, often replacing the entire range of an array by a window, i.e. few elements. When such array has to be communicated to the other units, only that window, i.e. few records at a time, can be sent out. Therefore program optimization causes a producer to store or send as few records at a time as feasible. Similarly, a consumer has also to store and consume a minimum number of records at a time. When producer and consumer processes are concurrent, the post and mail files require a buffer for a limited number of records. This type of data exchange realizes the concept of a pipeline or a stream. The user is not involved in this aspect of program design, however is warned if a file can not be exchanged in that fashion.

The units (processes) and files are the basic building blocks of a system in the MODEL environment. A system can be easily modified by composing a new configuration that includes existing, as well as new or modified, units and files.

The easy modifiability of a configuration supports several development modes. For example, individual units and files may be reused as the system is required to change. Entire independently developed systems may be easily interconnected by adding interfacing processes that convert commonly used variables from the form used in one system to that of the other. Thus, the creation of a new system that encompasses the functions of several old systems would not require designing of a new system.

Ada implementation:

Using Ada as an object language of the MODEL system gave us several advantages over using other high level languages. Ada multitasking and randezvous create a convenient tool for assembling parallel computations. Each MODEL specification is translated into a task. Configuration dependent parts of program units, like interconnections, are encapsulated into separately compiled subprograms and subtasks. A configuration unit, also generated by the configurator, assembles the parallel computation by simply enumerating in its body all the participating units with the 'WITH' clause. Only configurator generated parts of the overall computation have to be recompiled if the configuration changes.

In our design of an Ada implementation of the MODEL specifications, we stressed the independence of computation and configuration descriptions. Units generated by the MODEL compiler need to be compiled in Ada only once. The naming can be local in program units, and the configuration provides the translation of file names in different units. The user is able to select any set of such units and, after providing a configuration specification, generate a configuration unit that will run the entire computation. The configuration unit is compiled separately from MODEL units. Any change in configuration unit does not require MODEL units recompilation. Such solution provides high degree of modularity and supports easy assembling of new systems from existing computational units. Thus, it facilitates fast prototyping and bottom-up development and debugging of real-time systems.

The devised scheme of compilation is as follows:

Each mail file is replaced by a task. This task receives messages from producers, stores them in a queue and then, on the consumer request, moves them to the consumer. Sender and consumer establish randezvous with this task and not directly with each other. Due to the name independence (the same file can be named differently in different program units), sending messages is done through a re-router procedures which are generated by the Configurator. These procedures contain configuration sensative address tables.

The generated ADA units are as follows:

A. Each MODEL specification of a program unit is compiled by the MODEL compiler into a group of the following packages:

   2. Packages for each source mail file in the following format:

```
-- SMAILN - source mail file name
-- UNITN - name of the unit with SMAILN
package UNITN_SMAILN is
task UNITN_SMAILN_mbx is
     entry -- for receiving mail
     entry -- for sending mail
end UNITN_SMAILN_mbx;
end UNITN_SMAILN;
package body UNITN_SMAILN is
task body UNITN_SMAILN_mbx is
-- body of the mailbox (queue of messages)
end UNITN_SMAILN_mbx;
end UNITN_SMAILN;
```

   2. A package for a unit procedure with the following structure:

```
-- SMAILN - source mail file name
-- TMAILN - target mail/post file name
-- UNITN - program unit name
with SMAILN_UNITN;   -- repeat
-- for each source mail file in the unit
package UNITN is
procedure UNITN_prog;
end UNITN;
package body UNITN is
procedure UNITN_TMAILN_c is separate;
-- repeat for all post and mail files
procedure UNITN_prog is
   task UNITN_tsk;
   task body UNITN_tsk is
-- code of the MODEL program unit
   end UNITN_tsk;
 begin
   null;
 end UNITN_prog;
end UNITN;
```

B. The configurator produces the following configuration units:

   1. For each target post or mail file in the configuration it will generate the re-router in the form:

```
-- TMAILN - target mail/post file
-- UNITN - program unit name
with UNITN;  -- repeat
-- for all consumer units
separate(UNITN) -- name of program unit
                -- which contains this file
procedure UNITN_TMAILN_c is
begin
   -- a table of address translation and
   -- case on the value of the table address.
end UNITN_SMAILN_s;
```

   2. A *configuration unit for invoking the entire computation:*

```
-- CONFN is the name of the configuration
with UNITN_SMAILN; -- repeat
     -- for all source mail files
with UNITN_TMAILN  -- repeat
     -- for target mail & post files
with UNITN -- repeat
     -- for all program units
procedure CONFN is
   begin
     UNITN.UNITN_prog; -- repeat
          -- for all program units
   end CONFN;
```

All Ada compilation units are compiled in the following order: A1, A2, and B (order of B1 relative to B2 is irrelevant). Changes in B units affect only the changed package (therefore changing connections between program units and/or adding/deleting program units from configuration is easy and simple). It is worthwhile to note, that during compilation of a program unit no knowledge of configuration in which this unit will participate is needed.

4. MODEL COMPILER

The compilation of an equational specification into an object code consists of four stages: syntax analysis, semantic analysis and checking, scheduling of program events, and generation of the program. The later three stages, relevant to this paper, a summerized below.

Semantic Analysis and Checking:

The compiler translates the specification into a directed graph of data dependences. Use of data dependence graphs to optimize programs, in particular for parallel execution, has been proposed recently in the literature (see for example [Allen et al, 1983], [Ferrante, Ottenstein, and Warren 1984], [Kuck et al, 1981; Waters, 1983]). The distinctive feature of the array graph of the MODEL language is the compact representation of data dependences (a node represents entire array not a single element) and the lack of control dependences (flow of control is generated by the compiler).

Checking the specification and making corrections and additions may be regarded as inferring or propagating attributes from node to node. Thanks to nonprocedural semantics of the MODEL language we were able to implement powerful consistency checks in the compiler. Experience has shown that these checks are effective in locating 80-90% of the errors (not including syntax errors) in development of a program [Szymanski, et al, 1984].

Scheduling Program Events:

In composing a unit specification the user chooses natural and convenient data structures and equations. Typically this choice does not correspond to the most efficient implementation. In addition, the user description of data is independent of the medium of the data and whether it is internal (in main storage), external (secondary storage), or exchanged (communication line carrying messages). It is up to the compiler to map the user's specification into an efficient procedural computer program.

The optimization of the schedule proposed in the MODEL compiler is based on merging scopes of iterations to enable elements of the same or related structures to share memory locations. Usually there are many ways in which components can be merged (for different dimensions), each corresponding to different total orderings of the component graph. The memory requirements of different candidate scopes of iterations serves as the criterion for selecting the optimal merging and corresponding total ordering of the schedule. The selection is equivalent to NP-complete problem of finding a clique with the maximum weight of nodes in an undirected graph. Therefore a heuristic is used [Szymanski, 1987].

Generating Ada Code:

The final step of compilation is program generation that translates the individual entries in the schedule into the object code. In generating Ada code, the MODEL compiler heavily depends on the library of generic procedures for i/o conversions and mathematical operations. These generic procedures are differently instantiated in the generated programs according to the data types used in the specification. The object programs also use overloaded definitions of mathematical functions and operators to keep them independent of the used data types. The generated code use only standard features of Ada. It can be easily added to the existing Ada software. It can also be used as a part of the overall software development process.

5. CONCLUSION

The MODEL equational language provides the programmer with a powerful tool for very-high level, nonprocedural development of the executable system specifications. The MODEL compiler enables rapid prototyping and ensures high level of correctness and consistency checking. Ada, as an object code for the MODEL compiler, provides an efficient implementation tool for parallel execution of the equational specifications. It also ensures smooth synthesis of automatically generated Ada code with the existing Ada software.

Use of an equational language for expressing computations shields the user from considering low level implementation details, like describing input/output operations, loop structure, flow of control in the program etc. Compilation of specifications, including optimization and synchronization algorithms and customized code generators provides the user with efficient implementations of real time systems. Three cooperating components of the MODEL system: MODEL compiler, Configurator, and Timing Evaluator, constitute an integrated software development system that supports rapid prototyping, modularization and comprehensive consistency checking.

Acknowledgement. This work was partially supported by the Office of Naval Research under contract N00014-86-K-0442 and by Computer Command & Control Company. The author wish to express his thanks to Dr. Noah Prywes and Dr. Yuan Shi for discussions of the MODEL to Ada code generator.

References

1. ANSI/MIL-STD-1815 A, Reference Manual for the Ada Programming Language, DOD, Washington D.C., 1983.

2. Brinch, H.P., "Distributed Processes-A Concurrent Programming Concept", CACM, Vol. 21, No. 11, pp. 934-941, Nov. 1978.

3. Brinch, P.H. "EDISON - a Multiprocessor Language", Software-Practice and Experience, Vol. 11, pp. 325-361, 1981.

4. J. Ferrante, K.J. Ottenstein, and J.D. Warren, "The Program Dependence Graph and its Use in Optimization," Proc. 6th International Conference on Programming, LNCS, vol. 167, pp. 125-132, 1984.

5. Kuck, D.J., Kuhn, R.H., Padua, D.A., Leasure, B., and Wolfe, M., "Dependence Graphs and Compiler Optimizations," Proc. 8th ACM symp. Principles Programming Languages, Jan. 1981, pp. 207-218.

6. Lamport, L., "Specifying Concurrent Program Modules", ACM Trans. On Programming Languages and Systems, Vol. 5, No. 2, pp. 190-222, April, 1983.

7. Laner, P.E., Torrigiani, P.R. and Shields, M.W., "COSY - A System Specification Language Based On Paths and Processes," Acta Informatica Vol. 12, pp. 109-158, 1979.

8. I. Lee, N. Prywes and B. Szymanski, "Partitioning of Massive/Real-Time Programs for Parallel Processing," Advances in Computers, vol. 25, Academic Press, New York, 1986, pp. 215-275.

9. Martin, T., "Real Time Programming Language PEARL-Concept and Characteristics", Proc. COMPSAC, 1978.

10. Milner, R., A Calculus Of Communicating Systems, Lecture Notes on Computer Science, Vol. 92, Springer Verlag, 1980.

11. N.S. Prywes, B. Szymanski, and Y. Shi,"Very-High Level Concurrent Programming," IEEE Transactions on Software Engineering, vol. SE-13, no. 8, pp. 1038-1046, September, 1987.

12. Ramamrithan, K. and Keller, R.M., "Specification of Synchronizing Processes," IEEE Trans. On Software Engineering, Vol. SE-9, No. 6, pp. 722-733, Nov. 1983.

13. Szymanski, B., Lock, E., Pnueli, A., and Prywes, N.S., "On the Scope of Static Checking in Definitional Languages," Proceedings of the ACM Annual Conference, San Francisco, CA, 1984, pp. 197-207.

14. Szymanski, B., and Prywes, N., "Efficient Handling of Data Structures in Definitional Languages," Science of Computer Programming, 1987, to appear.

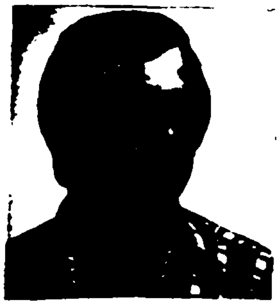
15. Teichreow, D. and Hershey III, E.A., "PSL/PSA: A Computerized Technique For Structured Documentation and Analysis of Information Processing Systems," IEEE Trans. on Software Engineering, Vol.SE-3, No.1, pp. 41-48, Jan. 1977.

16. J. Tseng, B. Szymanski, Y. Shi, and N.S. Prywes, "Real-Time Software Life Cycle with the MODEL System," IEEE Transactions on Software Engineering, vol. SE-12, No. 2, February, 1986, pp. 358-373.

17. R.C. Waters, "Expressional Loops," Proc. 10th ACM Symposium Principles of Programming Languages, ACM, 1983, pp. 1-10.

18. Wirth, N., "Toward a Discipline of Real-Time Programming", CACM, Vol. 20, No. 8, pp. 577-583, August 1977.

19. Zave, P., "An Operational Approach to Requirements Specification for Embedded Systems," IEEE Trans. On Software Engineering, Vol.SE-8, No.3, pp. 250-269, May, 1982.



Dr. Boleslaw Szymanski

Boleslaw Szymanski received Ph.D. in Computer Science from the Polish Academy of Science, Warsaw, Poland in 1976. In 1978 he was a Postdoctoral Fellow at Aberdeen University, Aberdeen, UK. He joined Rensselaer Polytechnic Institute in Troy, NY, as Associate Professor of Computer Science in 1985. Prior to that, he worked at Warsaw Technical University, Institute for Scientific, Technical, and Economic Information in Poland, and University of Pennsylvania, Philadelphia, PA.

Currently, his research interest includes very high level programming languages, parallel algorithms and systems, and large scale, distributed scientific computations. In the past he was also working on database system design and compiler construction and optimization.

Dr. Szymanski is an autnor of more than 50 scientific publications. He is a senior member of the IEEE Computer Society and member of the Association for Computing Machinery and the Mathematical Association of America.

END
DATE
FILMED
5-88
DTIC