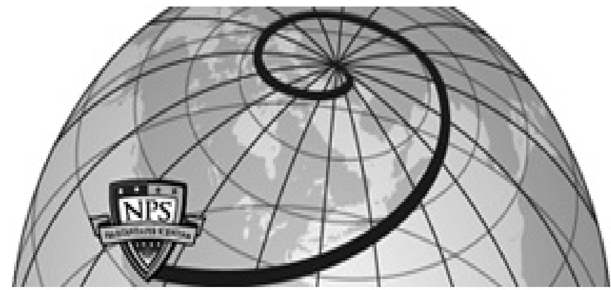
Calhoun
Institutional Archive of the Naval Postgraduate School



Calhoun: The NPS Institutional Archive

DSpace Repository

Faculty and Researchers | Faculty and Researchers' Publications

April 1988

Generating a Language Translator Based on an Attribute Grammar Tool

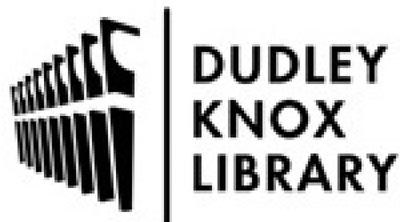
Herndon, Robert; Luqi

Naval Postgraduate School

R. Herndon and Luqi, "Generating a Language Translator Based on an Attribute Grammar Tool", Technical Report NPS 52-88-007, Computer Science Department, Naval Postgraduate School, 1988.

https://hdl.handle.net/10945/65245

Downloaded from NPS Archive: Calhoun



Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

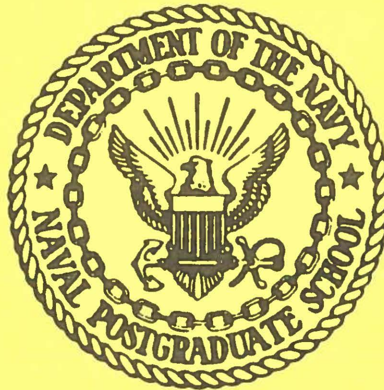
http://www.nps.edu/library

T 15

NPS52-88-007

NAVAL POSTGRADUATE SCHOOL

Monterey, California



GENERATING A LANGUAGE TRANSLATOR BASED ON
AN ATTRIBUTE GRAMMAR TOOL

LuQi

Robert Herndon

April 1988

Approved for public release; distribution is unlimited

Prepared for:

Naval Postgraduate School
Monterey, CA 93943

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R. C. Austin
Superintendent

K. T. Marshall
Acting Provost

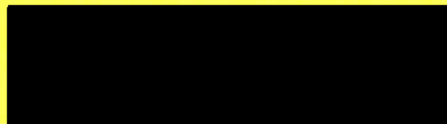
This report was prepared in conjunction with research conducted under the Naval Postgraduate School Foundation Research Program.

Reproduction of all or part of this report is authorized.

This report was prepared by:

LUQI
Associate Professor
of Computer Science

Reviewed by:



VINCENT Y. LUM
Chairman
Department of Computer Science

Released by:

JAMES M. FREMGEN
Acting Dean of Information
and Policy Science

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY 2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NPS52-88-007	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b. OFFICE SYMBOL (If applicable) 52	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943	7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Naval Postgraduate School 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
GENERATING A LANGUAGE TRANSLATOR BASED ON AN ATTRIBUTE GRAMMAR TOOL (U)

12. PERSONAL AUTHOR(S) LUQI, HERNDON, Robert

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
	FROM Oct 87 TO Mar 88	APRIL 1988	22

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	rapid prototyping, attribute grammars, translator generator, Ada, specification language, embedded system design, application generator, computer aided software engineering

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

A translator for the prototyping language PSDL is an important part of the associated computer-aided prototyping system. PSDL enables the prototype designer to rapidly construct prototypes of real-time systems by adapting and interconnecting reusable Ada software components. This paper describes the design of an experimental version of such a translator and its rapid implementation using an attribute grammar and a translator generation tool.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL LUQI	22b. TELEPHONE (Include Area Code) (408)646-2735 22c. OFFICE SYMBOL 52Lq

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted.
All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

☆ U.S. Government Printing Office: 1986—606-243

UNCLASSIFIED

Generating a Language Translator based on an Attribute Grammar Tool[1]

Luqi

Naval Postgraduate School Monterey, California 93943

Robert Herndon

University of Minnesota Mpls, MN 55455

ABSTRACT

KEYWORDS

rapid prototyping, attribute grammars, translator generator, Ada[2], specification language, embedded system design, application generator, computer aided software engineering

INTRODUCTION

We believe that computer aided prototyping shows promise. The Computer Aided Prototyping System (CAPS) system frees designers from many implementation details by constructing executable specifications from reusable components maintained by a software design management system [Luqi 1988a]. This system is particularly intended for the development and testing of real-time and control specifications and applications.

Our approach to rapid prototyping uses an prototyping language (PSDL - Prototype System Description Language) [Luqi 1988c] integrated with a set of software tools, including a user interfaces to speed up design entry and prevent syntax errors, an execution support system to demonstrate and measure prototype behavior and to perform static analyses of the prototype design, a software design-management system to manage reusable software components and design data, a software base to store reusable components, and a design database to store the prototype design. The prototyping language is an integral part of the CAPS software tools, since PSDL specifications are used by the software design management system for organizing and retrieving Ada reusable components in the software base. The prototyping language allows the

[1]This research was supported in part by the National Science Foundation under grant number CER-8710737.

[2]Ada is a registered trademark of the United States Government Ada Joint Program Office.

designer to use dataflow diagrams with non-procedural control constraints as part of the specification of a hierarchically structured prototype. The resulting description is free from programming level details, in contrast to prototypes constructed using a programming language. The underlying computational model unifies data flow and control flow, providing a vehicle suitable for developing top-down decompositions. Such decompositions allow large prototypes to be executed with practical computation times, in contrast to prototyping by simulating specifications via logic programming without providing a system architecture [Luqi 1988b].

PSDL is optimized for use at the specification and design level. The structure of the language encourages modular design of the prototype, and by extension the eventual production version. Special structures exist for describing real-time systems.

Two kinds of building blocks, representing **function** and **data abstractions** in PSDL, are used for constructing a PSDL prototype: operator and data type. A PSDL description represents a system decomposition as operators communicating via data streams. Each data stream carries values of a fixed abstract data type. Each data stream can also contain values of the built-in type "exception". The operators may be either data driven or periodic. Periodic operators have traditionally been the basis for most real-time system design, while the importance of data driven operators for real-time systems is recognized [Luqi 1986]. Such a description is based on the PSDL computational model. Formally the computational model is an augmented graph

$$G = (V, E, T(v), C(v))$$

where V is the set of vertices, E is the set of edges, T(v) is the maximum execution time for each vertex v, and C(v) is the set of control constraints for each vertex v. Each vertex is an **operator** and each edge is a **data stream.** The first three components of the graph are called the enhanced data flow diagram.

Abstractions are an important means for controlling complexity [Berzins 1988a]. This is especially important in rapid prototyping because a system must appear to be simple to be built or analyzed quickly. PSDL supports three kinds of abstractions: operator abstractions, data abstractions, and control abstractions. First two are mentioned in the previous paragraph. **Control abstractions** in PSDL are extracted by the computational model and are represented as enhanced data flow diagrams augmented by a set of non-procedural control constraints. Periodic execution is supported explicitly since it is a common property of

real-time systems. Conditional execution is supported by PSDL triggering conditions and conditional outputs.

Prototyping languages support executable specifications. There are two approaches to making a prototyping language executable, one based on meta-programming and the other on executable specifications [Berzins 1988b]. The PSDL prototyping language uses the first approach -- PSDL programs are specifications of the desired systems' behavior. It is an executable program whose behavior can be tested if all required information is supplied. The software base contains implementations for all atomic operators and types. The language uses the enhanced dataflow diagram as a basis for combining operators. Control constraints and timing constraints are the fundamental mechanisms to aid execution. Ada is used for implementing both the PSDL reusable components in the software base and the PSDL execution support environment.

EXECUTION SUPPORT SYSTEM

The PSDL Execution Support System [Luqi 1987] contains a static scheduler, a translator, and a dynamic scheduler. The static scheduler produces a static schedule for the operators with real-time constraints. The static schedule is a piece of Ada code containing calls to the program units implementing the operators to be scheduled. The translator augments the implementations of the atomic operators and types with Ada code realizing the data streams and activation conditions, resulting in an set of executable Ada programs for realizing the prototype system. Execution is under the control of the dynamic scheduler, which schedules the operators without real-time constraints. Figure 1 describes the functions of each component of the Execution Support System and the relationships among them.

The **translator** generates executable Ada code for PSDL operators and data streams [Moffitt 1988]. As a research tool, translator itself must be easily modifiable to reflect changes in PSDL or changes in the mapping from PSDL to Ada. This is accomplished by using an attribute grammar to define the translator at a high level, and an automated tool to generate the detailed code for parsing and translation. Experimentally, the translator generates Ada code to implement the operators as procedures which will be called by the Ada program generated by the Static Scheduler. One example of a change to the translation to be explored in future research is a buffer elimination optimization which would use Ada packages to imple-

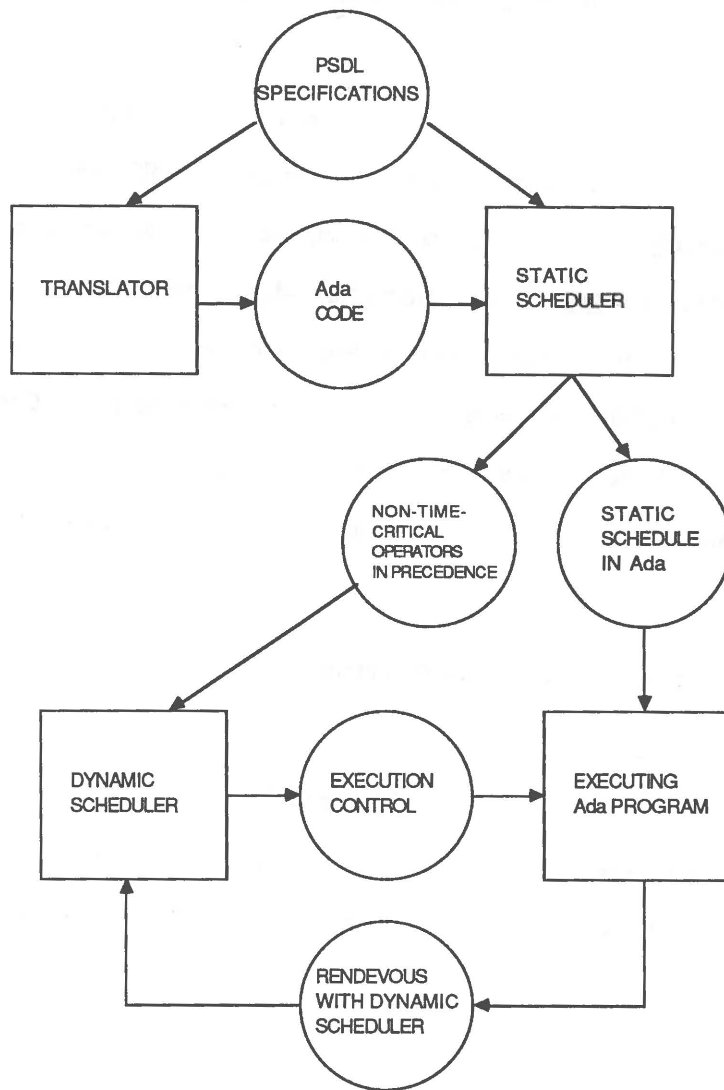


Figure 1. Execution Support System

ment PSDL state machines. The translator is responsible for instantiating generic packages which realize the data stream buffers between operators. The translator also ensures that all operator triggering conditions are encoded correctly, and that the Timer data type and the Exception data type are available if needed in the final model.

The **static scheduler** examines the PSDL source file to identify all operators having real-time constraints and to determine the precedence relations among the operators [Janson 1988, O'Hern 1988]. The static scheduler produces a schedule which takes into account the worst case time requirements for all

operators that have critical, real-time constraints such as a maximum execution time, minimum calling period, and minimum response time. This information is encoded into the Ada program to enforce timing constraints at run-time.

The **dynamic scheduler** operates at run-time along with the prototype model [Eaton 1988]. It is designed to control the execution of all non-critical operators within the program. A noncritical operator is one which is not subject to hard real-time constraints. The dynamic scheduler is invoked at run-time whenever there is spare time within the static schedule. The dynamic scheduler commences execution of the next available Ada program in its set of operators and continues to invoke non-critical modules until the available time is exhausted. At that point, operation of the dynamic scheduler is interrupted and control is returned to the static scheduler to continue the time critical operations.

MAPPING FROM PSDL CONSTRUCTS TO ADA

It is necessary to develop a mapping from the constructs of PSDL to the constructs of Ada in order to build a translator. The mapping is relatively straightforward because PSDL was designed to abstract and simplify the constructs of Ada relevant to the design of real-time systems. The semantics of some of the fundamental constructs of PSDL are explained below along with a possible implementation mapping into Ada. This mapping corresponds to an experimental version of the PSDL translator currently under development.

1. Operator

An **operator** is either a function or a state machine. When an operator fires, it reads one data object from each of its input streams, and writes at most one data object on each of its output streams. The output objects produced when a function fires depend only on the current set of input values. The output values produced when a state machine fires depend on the current set of input values and the current values of a finite number of internal state variables. Operators of these two types are useful for prototyping real-time systems.

Operators are either atomic or composite. Atomic operators cannot be decomposed in the PSDL computational model. Composite operators have realizations as data and control flow networks of lower level operators.

The Operator construction of PSDL is implemented by producing an Ada procedure. This procedure contains code to implement input and output to PSDL data streams, PSDL triggering conditions, and PSDL conditional output statements. Before presenting an example of this construction it will be necessary to describe the implementation of the PSDL data streams.

2. Data Streams

A **data stream** is a communication link connecting exactly two operators, a producer and a consumer. Communication links with more than two ends are realized using copy and merge operators. Each stream carries a sequence of data values.

There are two types of data streams - **dataflow** streams and **sampled** streams. A dataflow stream guarantees that none of the data values are lost or replicated, while a sampled stream guarantees the most recently generated data value is always available. Dataflow streams are used to control operators subject to the natural dataflow firing rule. Sampled streams are used to connect operators that fire at incompatible frequencies.

A PSDL stream is mapped into a buffer capable of holding one data value. Since a buffer may be read by an operator executing independently of the operator writing into the buffer, they must be protected from data conflicts due to concurrent access. Consequently buffers are embedded in Ada tasks and read or written via task entries, to provide mutually exclusive access. The buffer manager task can be declared inside a generic package to make it easy for the translator to create a separate buffer manager task for each PSDL data stream. Thus each PSDL data stream is implemented by an instance of a generic package.

Two kinds of buffers are needed, corresponding to the two kinds of data streams in PSDL. Sampled buffers are used to implement sampled streams and FIFO buffers are used to implement dataflow streams. The difference between the two kinds of buffers is that a FIFO buffer makes sure that every value written into the buffer is read exactly once before the next value is written into the buffer. Violations of this constraint are reported via Ada exception conditions. There are two possible exceptions: Underflow and Overflow. Underflow is raised if the consumer operator attempts to read the buffer before it has been updated by the producer operator. Overflow is raised if the producer attempts to write to the buffer before the consumer has read the previous data value. There are no constraints on the order a sampled buffer is

accessed, and no associated exception conditions.

3. Data Triggers

The translator must select the appropriate type for buffer for a given data stream according to the triggering conditions of the consumer operator associated with the stream. There are two types of **data triggers** for PSDL operators.

OPERATOR p TRIGGERED BY ALL x, y, z

OPERATOR q TRIGGERED BY SOME a, b

In the first example the operator p is ready to fire whenever new data values have arrived on all three of the input arcs x, y, and z. This rule is a generalized form of the natural dataflow firing rule since in PSDL a proper subset of the input arcs can determine the triggering condition for an operator, without requiring new data on all input arcs. This kind of data trigger can be used to ensure that the output of the operator is always based on fresh data for all of the inputs in the list, and can be used to synchronize the processing of corresponding input values from several input streams.

In the second example, the operator q fires when any of the inputs a and b gets a new value. This kind of activation condition guarantees that the output of operator q is based on the most recent values of the critical inputs a and b mentioned in the activation condition for q. If q has some other input c, the output of q can be based on old values of c, since q will not be triggered on a new value of c until after a new value for a or b arrives. This kind of trigger can be used to keep software estimates of sensor data up to date.

If an operator mentions a data stream in a TRIGGERED BY ALL condition then a FIFO buffer will be selected for the stream. If the operator mentions a data stream in a TRIGGERED BY SOME condition then a sampled buffer will be selected for the data stream. If an operator does not mention an input stream in either kind of triggering condition than a sampled buffer will be selected also. It is illegal for an operator to mention the same data stream in both a TRIGGERED BY ALL and a TRIGGERED BY SOME condition, since this would introduce an inconsistency. For example, in Figure 2, operator T has four input streams. The specification for T is TRIGGERED BY ALL D, F, H. The translator will select FIFO buffers

for streams D, F, and H. Stream G will be a sampled buffer. In the same figure, operator P has four input streams. The specification for P is TRIGGERED BY SOME R. In this case all data streams will be sampled. Again in Figure 2, operator FF has two input streams. The specification for FF lacks a TRIGGERED BY token, so that both streams will be sampled. The triggering conditions for the consumer operator determine the type of each data stream.

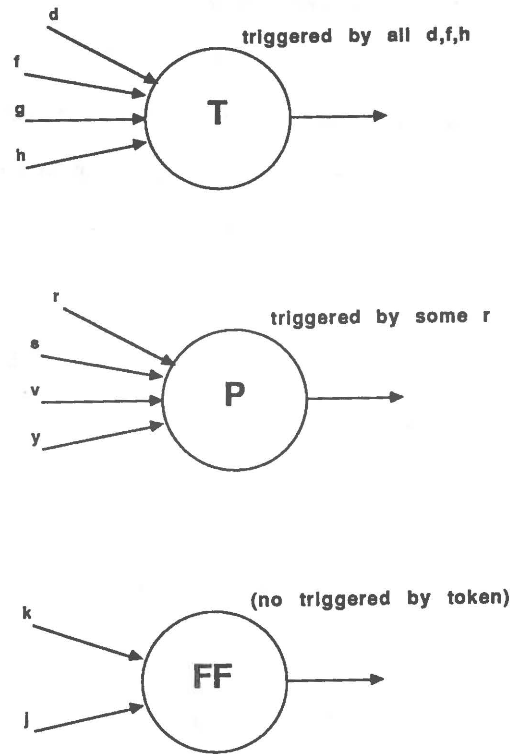


Figure 2. Queue selection based on the "TRIGGERED BY" construct

There is one more issue related to buffer management that must be addressed. The static scheduler generates a schedule for the time critical operators and this schedule is enforced to ensure real-time constraints are met. Some operators do not have time critical constraints. These operators are executed during unused time periods in the schedule for the time critical operators. It is possible that a time critical operator is the consumer of a data stream produced by a non time-critical operator. The time critical operator has priority and is scheduled to run by the static scheduler in some repetitive cycle. The non time-critical operator is fired at the convenience of the dynamic scheduler, in the excess time in the main schedule. A non time-critical operator can be attempting to write to a data stream when it is interrupted by the dynamic scheduler in order to run a time critical operator. If the time critical operator is the consumer for the data from the non time-critical operator, it may be unable to read a value from the buffer because the buffer task is not ready to execute an accept statement, and is blocked because the execution of the non time-critical operator has been preempted.

This difficulty can be overcome by giving the buffer tasks an even higher priority than the time-critical operators. In this way, once the buffer task is called, whatever operation is taking place on the buffer must be allowed to complete before an interrupt can take place, since the priority associated with a rendezvous is the higher of the priorities of the two tasks involved. This will guarantee that no process can hang up the system by getting blocked in the middle of a buffer operation. The operation time for any buffer task is be very short, so there should be little time penalty in switching between tasks. Figures 3 and 4 contain Ada code implementing the two types of buffer tasks, SAMPLED STREAM and FIFO.

The get and put operations of the buffers are for reading and writing the values in the buffer. The check operations are used in the implementation of TRIGGERED BY ALL and TRIGGERED BY SOME to determine if a new data element is present in the buffers. These checks are performed each time an operator with a data trigger is scheduled to execute. If the triggering condition evaluates to false, the operator is not fired and control is passed to the dynamic scheduler to utilize the unused time slot. 4. Buffer Selection Conflicts

A problem which arises in buffer selection is illustrated in Figure 5. In this case a high level operator is decomposed into three lower level operators. The designer will enter a specification for both the top level operator A and for the lower level operators BB, CC, and DD. Suppose operator A is TRIGGERED

```
generic type ELEMENT_TYPE is private;
package SAMPLED is
  task SAMPLED_BUFFER is
    entry CHECK (NEW_DATA : out BOOLEAN);
    entry PUT (VALUE : in ELEMENT_TYPE);
    entry GET (VALUE : out ELEMENT_TYPE);
  end SAMPLED_BUFFER;
end SAMPLED;

package body SAMPLED is
  task body SAMPLED_BUFFER is
    BUFFER : ELEMENT_TYPE;
    VALUE : ELEMENT_TYPE;
    NEW_DATA_VALUE: BOOLEAN := false;
  begin
    loop
      select
        accept CHECK (NEW_DATA : out BOOLEAN) do
          NEW_DATA := NEW_DATA_VALUE;
        end CHECK;
      or
        accept GET (VALUE : out ELEMENT_TYPE) do
          VALUE := BUFFER; NEW_DATA_VALUE := false;
        end GET;
      or
        accept PUT (VALUE : in ELEMENT_TYPE) do
          BUFFER := VALUE; NEW_DATA_VALUE := true;
        end PUT;
      end select;
    end loop;
  end SAMPLED_BUFFER;
end SAMPLED;
```

Figure 3. A Sampled Stream Buffer Task

BY ALL a. Also suppose that operator BB does not contain the TRIGGERED BY ALL tokens. When the translator selects a buffer task for a, it will instantiate a FIFO buffer task to implement a. For BB, it would select a sampled stream task to implement a'. Although a and a' carry the same data, they have not been implemented with the same type of buffer. The translator does not presently check inheritance rules. In operation data would be placed onto a and would then be passed to a' and into BB. The user must prevent this type of error by ensuring that lower level operators which result from the decomposition of a higher level operator have the same triggering conditions at the input in order to prevent the buffer mismatch just demonstrated. This difficulty arises only for lower level operators which read the same input streams as the higher level operator. This is true because the type of buffer required at any point in the system is determined by the triggering conditions of a consumer operator. Therefore, decomposition rules do not affect

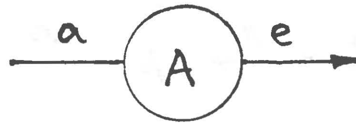
```
generic type ELEMENT_TYPE is private;
package FIFO is
  task FIFO_BUFFER is
    entry CHECK (NEW_DATA : out BOOLEAN);
    entry PUT (VALUE : in ELEMENT_TYPE);
    entry GET (VALUE : out ELEMENT_TYPE);
  end FIFO_BUFFER;
  BUFFER_READ_ERROR, BUFFER_WRITE_ERROR : exception;
end FIFO;

package body FIFO is
  task body FIFO_BUFFER is
    BUFFER : ELEMENT_TYPE;
    VALUE : ELEMENT_TYPE;
    NEW_DATA_VALUE: BOOLEAN := false;
  begin
    loop
      select
        accept CHECK (NEW_DATA : out BOOLEAN) do
          NEW_DATA := NEW_DATA_VALUE;
        end CHECK;
      or
        accept GET (VALUE : out ELEMENT_TYPE) do
          if NEW_DATA_VALUE then
            VALUE := BUFFER; NEW_DATA_VALUE := false;
          else raise BUFFER_READ_ERROR; end if;
        end GET;
      or
        accept PUT (VALUE : in ELEMENT_TYPE) do
          if not NEW_DATA_VALUE then
            BUFFER := VALUE; NEW_DATA_VALUE := true;
          else raise BUFFER_WRITE_ERROR; end if;
        end PUT;
      end select;
    end loop;
  end FIFO_BUFFER;
end FIFO;
```

Figure 4. A FIFO Buffer Task

the specification requirements of operators CC and DD in Figure 5. However, if A is TRIGGERED BY ALL a, then BB must be TRIGGERED BY ALL a'. Presently the translator does not enforce this constraint, although refinements addressing this issue are planned.

5. The State Buffers

A final issue in data stream implementation is initializing PSDL state variables to the values designated by the associated PSDL declaration STATES INITIALLY. Each state variable will have its own



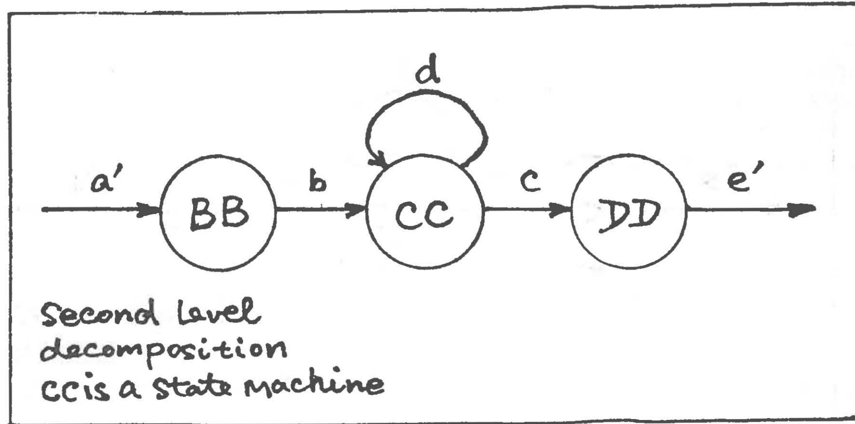


Figure 5. Stream Type Inheritance

buffer task. An example is shown in Figure 5. Operator CC is a state machine. It has a state variable which is transmitted along buffer task D. The data value traveling along D must have some initial value. That value is found in the STATES INITIALLY statement in PSDL. To insure the correct initial value for the state variables in the program, buffer task D must be loaded with the correct value prior running the prototype. An Ada procedure called PRELOAD is produced by the translator for all PSDL prototypes, containing a series of calls on the "put" entries of the buffer tasks to put the correct initial values into the appropriate buffers. If there are no state variables in the program, the procedure is empty. The static scheduler must call PRELOAD before the execution of any schedule it creates for the prototype. The preloading procedure is not part of the scheduler proper. It is run once to initialize the state buffers and is not run again unless the prototype program is restarted from the beginning.

IMPLEMENTATION OF THE TRANSLATOR

The translator is created using an automated translator generator called Kodiyak. Kodiyak was developed at the University of Minnesota [Herndon 1985, Herndon 1988a]. It is available as a research tool and is quite effective. The system is based on Knuth's attribute grammars [Knuth 1968]. It utilizes a variation of Jalili's algorithm [Jalili 1983] to evaluate the semantic tree it creates when generating a translation. More details about generating translators using attribute grammars can be found in [Reps 1983].

Producing a translator with Kodiyak requires a translator description consisting of three major components. This translator description is a formal description of the terminal and non-terminal tokens of the source language to be translated, a list of the attributes each token possesses, and any precedence relationships that may be required to properly evaluate ambiguous cases in the grammar. Finally, the file contains a set of attribute equations that describe the relationship between the source language (in this case PSDL) and the target language (in this case Ada). The translator generator system, Kodiyak, utilizes these equations to produce C, Yacc, and Lex specifications that are compiled to produce an executable translator. By running this program with a text file in the source language (PSDL) as input, an output file is created which contains the derived code in the target language (Ada).

As noted earlier, the translator is generated with an automatic translator generator called Kodiyak. The translator implementation is straightforward. Figure 6 illustrates the process.

Once the executable translator is created, it can be given any source program in PSDL and will output a source program in Ada. The essential difficulty is to specify the mapping between PSDL and Ada, such that the results of translation will be correct, compilable Ada code which will faithfully implement the system described in PSDL in Ada.

The translator generated by Kodiyak is capable of scanning an input file written in a specific language, parsing it, locating syntax errors, and if no errors are present, producing a translation in an output text file. The text file output can be a source program in another language.

In the present case the input language is PSDL and the output language is Ada. Kodiyak requires a description of the syntax of PSDL together with attribute equations. The attribute equations map PSDL constructs to the constructs of the target language Ada according to the mappings derived in the previous section.

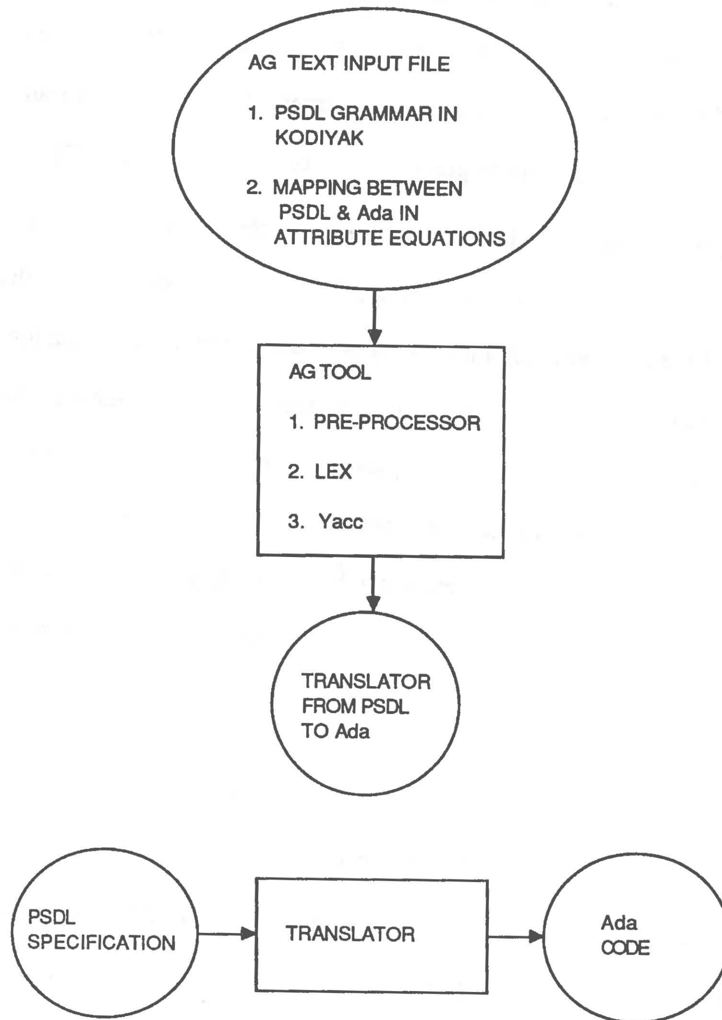


Figure 6.

Kodiyak is an attribute grammar based tool that generates executable translators. Particular translators are capable of scanning an input file written in a specific language, parsing it, locating any syntax errors, and if no errors are present, producing a translation in the form of an output text file. The text file output can then be compiled, linked, and exported to the operating system. In the present case the input language is PSDL and the output language is Ada. Kodiyak requires a description of the syntax of PSDL in attribute grammar form. The BNF grammar of PSDL must be related to the target language Ada. Figure 7 illustrates the various portions of the input file for the PSDL to Ada translator.

The first section of any Kodiyak file is the Lexical Definition section. In this section all the lexical symbols which make up the input language are defined. The %define line allows the definition of a variable name, e.g., DIGIT, which can be used in subsequent lexical definitions. All terminal symbols in the language are defined in this section. These are the tokens which the translator will expect to detect in the input file.

The next section of the Kodiyak file is the attribute declarations section. Here the attributes which non-terminal and terminal symbols may have are identified. In Figure 7 the non-terminal, operator_spec, is

Lexical Definition Section:

Terminal Symbols

```
%define ALPHA  :[a_zA_Z]
%define DIGIT  :[0-9]
OPERATOR      :operator|OPERATOR
MAX_EXEC_TIME  :maximum execution time|MAXIMUM EXECUTION TIME
```

Attribute Declarations Section:

Grammar Symbols *Attributes*

```
operator_spec        {trn:   string; };
max_exec_time        {trn:   string; };
ID                   {%text: string; };
```

Attribute Grammar Section:

Grammar Symbols *Attribute Equations*

```
time
    :NUMBER unit
      {time.trn = [NUMBER.%text,unit.trn]; }
    ;

unit
    :MICROSEC
      {unit.trn = ""; }
    |MS
      {unit.trn = "000"; }
    ;
```

Figure 7. Sample From the Kodiyak Input File Specification For translator

assigned one attribute, trn, which is of type string. Kodiyak allows attributes to be either string, integer, or map types. All attributes of the present translator are of type string since the objective is to convert a text file of PSDL into a text file of Ada.

Attributes of terminal symbols are restricted. In Figure 7, the terminal ID has the attribute, %text. This is a special attribute for terminal symbols, identified in Kodiyak by the % marker. It is a string-valued attribute and is initialized to the text matched by the terminal symbol.

The final portion of the Kodiyak input file is the attribute grammar itself. Here the syntax and semantics of the translation are specified. The BNF rules of the PSDL are defined and each is associated with an equation defining the attributes and their relation to the target language.

The brief discussion here does not reveal the full power and capabilities of Kodiyak. However, it should illustrate that Kodiyak provides simple yet effective means to accomplish translator generation.

CONCLUSION AND FUTURE RESEARCH

Currently the CAPS system is under development as a set of separate components. Conceptual work has been completed for the design of both the Static and Dynamic Schedulers. Implementation of the conceptual designs must be undertaken. The feasibility of the Translator has been demonstrated empirically. The Translator requires a rigorous, formal definition of the relationship between PSDL and Ada syntax. This definition must then be applied to the attribute equations in the Translator to achieve general applicability and the fullest use of PSDL's and Ada's capabilities.

The present version of Kodiyak used to generate the translator is an excellent tool. It generates an effective easily modified translator. However, some improvement in the error messages returned to the user when the translator is applied to a syntactically incorrect input file is needed, and no semantic error checking is performed. As the system develops, syntactic error recovery rules and additional attributes and attribute equations for semantic checks will have to be added to improve the robustness of the PSDL translator and prototypes.

Efforts to integrate the various parts of CAPS are being deferred as development proceeds on remaining portions of the system. At present work has commenced on the Software Base management

System at the conceptual and, to a limited degree, the empirical levels. Work is underway to develop the syntax directed editor for the system and portions of the graphic interface. As the remaining portions of the system are developed work will be required to integrate all the individual tools into an integrated work environment.

Automated facilities to translate a prototyping language into an underlying implementation language are feasible, and are a working reality at this time. Much work remains to develop the formal basis for the concept. The aims of this research effort are to create a sound foundation for the development and use of highly automated program development environments. Such environments are designed to improve productivity in software development and are a first small step on the way to fully automatic generation of complete programs from specifications. Before progress toward that end can be made, solid theory and empirical demonstration of principles in several areas must be made.

REFERENCES

[Berzins 1988a]
V. Berzins, Luqi, *Software Engineering with Abstractions: An Integrated Approach to Software Development using Ada*, Addison-Wesley, 1988.

[Berzins 1988b]
V. Berzins, Luqi, *Languages for Specification, Design and Prototyping*, Chapter in *Handbook of Computer-Aided Software Engineering*, Van Nostrand Reinhold, 1988.

[Eaton 1988]
S. Eaton, *An Implementation Design of a Dynamic Scheduler for a Computer Aided Prototyping System*, M. S. Thesis, Naval Postgraduate School, March, 1988.

[Herndon 1985]
R. Herndon, *The Incomplete AG User's Guide and Reference Manual*, University of Minnesota, Computer Science, Technical Report 85-37, October 1985.

[Herndon 1988a]
R. Herndon, and V. Berzins, *AG: A Useful Attribute Grammar Translator Generator*, to appear in IEEE TSE 1988. Also Tech. Report 85-25, Computer Science Department, University of Minnesota, 1985.

[Herndon 1988b]
R. Herndon, *Attribute Grammar Systems for Prototyping Translators and Languages*, Ph.D. dissertation, University of Minnesota, 1988.

[Jalili 1983]
F. Jalili, *A General Linear-Time Evaluator for Attribute Grammars*, ACM *SIGPLAN Notices*, Vol. 18, No. 9, September 1983, pp. 35-44.

[Janson 1988]
D. Janson, *A Static Scheduler for Hard Real-Time Constraints in the Computer Aided Prototyping System*, M. S. Thesis, Naval Postgraduate School, March, 1988.

[Knuth 1968]
D. Knuth, *Semantics of Context-Free Languages, Math. Syst. Theory* Vol. 2, No. 2, June 1968, pp. 127-145.

[Luqi 1986]
Luqi, *Rapid Prototyping for Large Software System Design*, Ph.D. dissertation, University of Minnesota, 1986.

[Luqi 1987]
Luqi, *Execution of Real-Time Prototypes, ACM First International Workshop on Computer-Aided Software Engineering*, Cambridge, Massachusetts, May 1987, Vol. 2, pp. 870-884.

[Luqi 1988a]
Luqi, M. Ketabchi, "A Computer Aided Prototyping System," *IEEE Software*, March 1988, pp. 66-72.

[Luqi 1988b]
Luqi, *Specification Languages in Computer Aided Software Engineering*, to appear in *Proceedings of IEEE Systems Design and Network Conference*, Santa Clara, CA, April, 1988.

[Luqi 1988c]
Luqi, V. Berzins, R. Yeh, *A Computer Aided Prototyping System*, to appear in *IEEE TSE, 1988*. Also Technical Report 86-4, Computer Science Department, University of Minnesota, Jan. 1986.

[Moffitt 1988]
C. Moffitt, II, *A Language Translator for a Computer Aided Rapid Prototyping System*, M. S. Thesis, Naval Postgraduate School, March, 1988.

[O'Hern 1988]
J. O'Hern, *A Conceptual Design of a Static Scheduler for Hard Real-Time Systems*, M. S. Thesis, Naval Postgraduate School, March, 1988.

[Reps 1983]
T. Reps, *Generating Language Based Environments*, Ph. D. Thesis, University of Massachusetts, Amherst, 1983.

Initial Distribution List

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93943	2
Center for Naval Analysis 4401 Ford Avenue Alexandria, VA 22302-0268	1
Office of the Chief of Naval Operations Code OP-941 Washington, D.C. 20350	2
Office of the Chief of Naval Operations Code OP-945 Washington, D.C. 20340	2
Commander Naval Telecommunications Command Naval Telecommunications Command Headquarters 4401 Massachusetts Avenue NW Washington, D.C. 20390-5290	2
Commander Naval Data Automation Command Washington Navy Yard Washington, D.C. 20374-1662	1
Office of Naval Research Office of the Chief of Naval Research Attn. CDR Michael Gehl, Code 1224 Arlington, VA 22217-5000	1
Director, Naval Telecommunications System Integration Center NAVCOMMUNIT Washington Washington, D.C. 20363-5100	1
Space and Naval Warfare Systems Command Attn: Dr. Knudsen, Code PD50 Washington, D.C. 20363-5100	1

Address	Copies
Ada Joint Program Office OUSDRE(R&AT) The Pentagon Washington, D.C. 230301	1
Naval Sea Systems Command Attn: CAPT Joel Crandall National Center #2, Suite 7N06 Washington, D.C. 22202	1
Office of the Secretary of Defense Attn: CDR Barber The Star Program Washington, D.C. 20301	1
Naval Ocean Systems Center Attn: Linwood Sutton, Code 423 San Diego, CA 92152-5000	1
Director of Research Administration Code 012 Naval Postgraduate School Monterey, CA 93943	1
Chairman, Code 52 Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	1
LuQi Code 52Lq Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	150