1988

Automated Translation from a Prototyping Language into Ada

Luqi; Moffitt, C. II

Naval Postgraduate School

Luqi and C. Moffitt II, "Automated Translation from a Prototyping Language into Ada", Technical Report NPS 52-88-009, Computer Science Department, Naval Postgraduate School, 1988.

https://hdl.handle.net/10945/65221

T 47

NPS52-88-009

NAVAL POSTGRADUATE SCHOOL
Monterey, California

AUTOMATED TRANSLATION FROM A PROTOTYPING LANGUAGE INTO ADA

C. Moffitt, II

LuQi

April 1988

Prepared for:

National Science Foundation
Washington, DC 20550

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R. C. Austin
Superintendent

K. T. Marshall
Acting Provost

This report was prepared for the Naval Postgraduate School. The work reported herein was supported in part by the National Science Foundation.

This report was prepared by:

[redacted]

LUQI
Associate Professor
of Computer Science

Reviewed by:

[redacted]

VINCENT Y. LUM
Chairman
Department of Computer Science

Released by:

[redacted]

JAMES M. FREMGEN
Acting Dean of Information
and Policy Science

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NPS52-88-009	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
6c. ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943	7b. ADDRESS (*City, State, and ZIP Code*)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION National Science Foundation 8b. OFFICE SYMBOL (*If applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER Agreement No. CCR-8710737 dated 27 Jul 87
8c. ADDRESS (*City, State, and ZIP Code*) Washington, D.C. 20550	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (*Include Security Classification*)
AUTOMATED TRANSLATION FROM A PROTOTYPING LANGUAGE INTO ADA (U)

12. PERSONAL AUTHOR(S) MOFFITT, C. II and LUQI

13a. TYPE OF REPORT	13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (*Year, Month, Day*)	15. PAGE COUNT
		APRIL 1988	

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, Ada, specification, language translator, computer aided software engineering

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

Rapid prototyping with a computer aided software prototyping system automates an important part of the development effort. Designers develop software prototypes in Prototype System Description Language (PSDL) at specification level. The automated translation described in this paper provides the mapping from PSDL to Ada and generates a translator for the execution.

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

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

AUTOMATED TRANSLATION FROM A PROTOTYPING LANGUAGE INTO ADA[1]

C. Moffitt, II

Luqi

Naval Postgraduate School
Monterey, California 93943

ABSTRACT

Rapid prototyping with a computer aided software prototyping system automates an important part of the development effort. Designers develop software prototypes in Prototype System Description Language (PSDL) at specification level. The automated translation described in this paper provides the mapping from PSDL to Ada and generates a translator for the execution.[2]

KEYWORDS

rapid prototyping, attribute grammars, Ada, specification, language translator, computer aided software engineering

1. A Computer Aided Prototyping System (CAPS)

A CAPS has been proposed in "Rapid Prototyping for Large Software System Design" [Luqi, 1986]. The central objective of the system is to optimize the use of the programmer's time and improve the quality of developed prototypes [Luqi Berzins, 1988]. The objective of prototype development is to:

(1) provide a firm set of requirements and functional specifications which will guide development of the production software.
(2) ensure agreement between customer and developer as to the requirements and expected performance characteristics of the system
(3) generate a modular, skeletal structure of the software system which will serve to guide further implementation
(4) shorten prototype development time and thus accelerate production system delivery
(5) assist in estimating the ultimate development costs of the finished system

The CAPS architecture consists of a set of major subsystems [Luqi Ketabchi, 1988]. Each subsystem complements the others and forms a powerful tool for prototyping large and real-time software systems. Figure 1 illustrates the relationship of the various subsystems.

---

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

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

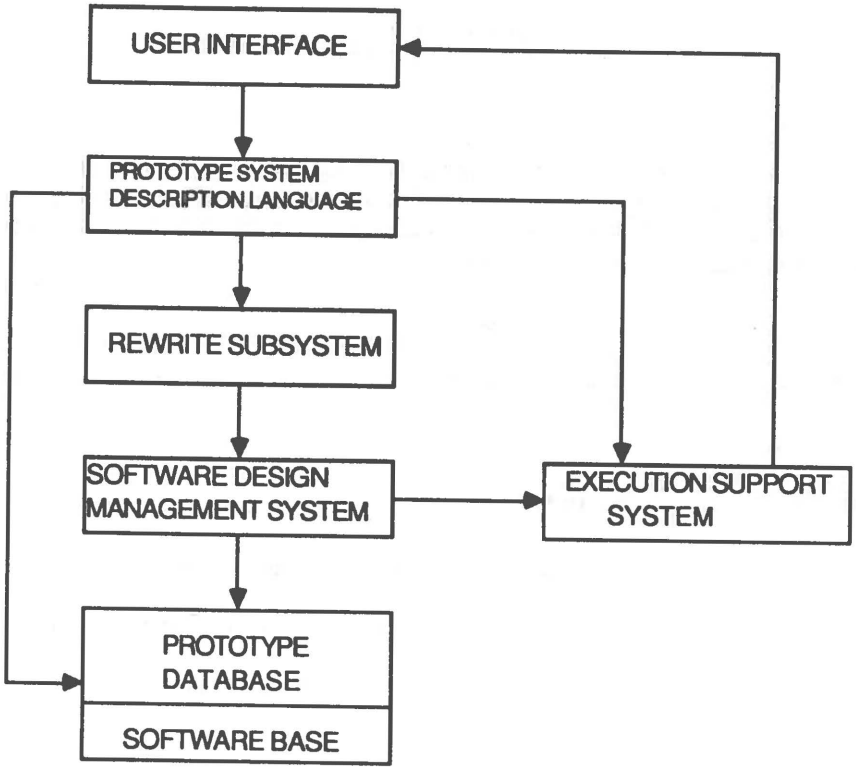

Figure 1. The overall structure of CAPS

2. "Prototype System Description Language (PSDL)"

The core of the CAPS is the Prototype System Description Language (PSDL) [Luqi Berzins Yeh, 1988]. It is optimized for use at the specification and design level[Luqi, 1988]. PSDL allows the user to describe the system in terms of its required or desired performance characteristics. Much of the detail of implementation in a programming language is abstracted away. The questions emphasized are, what does the system do?; and what performance parameters must be met? Special language constructs exist for describing real-time systems. This data-flow based language encourages modular design of the prototype, and by extension the eventual production version of the software system.

PSDL provides two kinds of system components, operator and data stream, for constructing software prototypes.

(1) Operators

PSDL models a software system as a set of OPERATORS communicating via DATA STREAMS [Luqi, 1986]. In PSDL, Operators may be either atomic or composite. Composite operators can be decomposed into two or more operators, each of which may be composite or atomic. Atomic operators cannot be further decomposed. PSDL envisions a hierarchical breakdown of the system into logical components which are as simple as possible without becoming trivial. No special rules for decomposition are imposed. This distinction allows the modeling of hierarchically structured programs as sets of operators. Operators at higher levels in the program structure are composite while those at the lowest level of the program structure are the atomic operators. PSDL can therefore be used to support top down design strategies.

Operators may be either data driven or periodic. Under this schema, the firing of a data driven operator is accomplished due to the presence of data in its' input data stream(s), while the firing of a periodic operator is dependent upon timing constraints which must be met during program operation. The data driven operator allows the modeling of systems which utilize data flow as a means of control vice the more traditional timing control in real-time systems. In either case, when an operator fires, it reads one data object from each of its' input streams and writes, at most, on object to each of its' output streams [Luqi, 1986].

A third classification of operators is allowed. An operator may be either a function or a state machine. This description relates to the values output from the operator. The output value of the function type operator is dependent solely on the current set of values present on the input streams to the operator. The output of the state machine type depends, not only upon the current set of input values, but also on the values of a finite number of state variables internal to the operator.

(2) Data Streams

In PSDL, data streams represent a communication link between exactly two operators. One operator is the producer of the data while the other is the consumer of the data. There are two types of PSDL data streams. One is the is a DATA FLOW STREAM the other is the SAMPLED STREAM. The DATA

FLOW STREAM can be thought of as a first in first out (FIFO) queue capable of holding, at most, one data value. This data value may be used one time by the consumer operator. It may not be overwritten by the producer. In effect, this stream guarantees deliver of the data value, and guarantees that each individual data value will be read once and only once. The second type queue can also be thought of as a queue of length one. In this case, however, the data value may be overwritten by the producer before the consumer reads it, or may be read multiple times by the consumer, or not at all. In the sampled stream case, delivery of an individual data value is not guaranteed. The choice of data stream is dependent upon the control conditions specified for the operator.

2.1. "Operator Control in PSDL"

Two types of control are allowed in PSDL. The first is periodic. This is a common form of operator control in which operators are fired by some regular schedule. Periodic control is supported in PSDL by several constructs. The primary construct is PERIOD followed by a time value. Period implies that the operator must fire sometime between the beginning of the period and some deadline which defaults to the end of the period. Thus, PERIOD is an upper bound on the length of time allowed between any two firings of a given operator. This is an explicit period.

The second type of operator is sporadic. Sporadic operators represent the second form of control allowed in PSDL. Sporadic operators are triggered by the arrival of data on the input streams of an operator (Luqi, 1986). This form of control is what is referred to as Natural Data Flow (NDF) in the PSDL schema. It is dependent on the flow of data through the prototype to cause the firing of operators. However, timing control may be introduced to this form of control through the use of special PSDL tokens. These tokens are:

MAXIMUM EXECUTION TIME (MET)
MAXIMUM RESPONSE TIME (MRT)
MINIMUM CALLING PERIOD (MCP)

A time value would be specified with each of the above tokens. These three language structures are a major means of specifying real-time constraints for a system in PSDL. MET is an upper bound on the length of time which may elapse from the beginning of execution of a module to the end of the execution of that module. MRT has two different interpretations. The first applies to periodic operators. In this case,

MRT is an upper bound on the time from the beginning of a *period* and the time when the last data has been output onto the output stream of the operator. The second case for MRT applies to Sporadic operators. For the Sporadic operator, MRT is an upper bound on the elapsed time from the arrival of new data on the input streams to the operator and the time when the last data value is placed on the output stream of the operator. MCP is a lower bound on the elapsed time allowed between the arrival of one set of values on the input streams of an operator and the arrival of the next set of values on the input streams. All operators may have MET. Sporadic operators may have MRT and MCP. If a sporadic operator has an MRT it must also have an MCP [Luqi, 1986].

NDF control of sporadic operators is signified by the PSDL token TRIGGERED BY. This token will be qualified by either the additional token ALL or SOME. TRIGGERED BY ALL indicates that an operator is to be fired when new data values have arrived on all the input streams to the operator. TRIGGERED BY SOME implies that the operator will be fired by the arrival of a new data value on any one (or possibly all) of the input streams to the operator. The designer must specify which input streams the TRIGGERED BY ALL/SOME construction refers to. He may specify a proper subset of the input streams in either case. In this way, if an operator has multiple input streams, but only a few of them are critical to the firing of the operator, the designer may so specify. Conditional firing of operators can be accomplished by the addition of input or output predicates in the PSDL specification.

2.2. Timer

TIMER is a PSDL construct which is useful in the development of real-time systems. A timer is an abstract state machine. In PSDL it is somewhat like a stopwatch. It has the primitive operations of START, STOP, RUN, and RESET. It is used for such things as measuring the length of time between two events, or the length of time the system or an operator has remained in a particular state. In the prototype CAPS, TIMER does not function in the same way as a clock construct for an operating system. It does not provide normally direct control of operator firing. It can be used as a value for a PSDL input or output conditional to act as a guard to the firing of an operator. It is primarily provided to collect statistics about the prototype system.

2.3. Exception

PSDL supports both normal and EXCEPTION data types. The PSDL EXCEPTION is a built in type. It can be transmitted on any data stream as a data value. It can be suppressed by the use of input or output conditionals. It can be handled in PSDL or in Ada. Some possible operations for the PSDL EXCEPTION are:

(1) to create an exception with a given name
(2) to detect if a value on a data stream is:
    a. an exception with a given name
    b. normal (not an exception)

3. Execution Support System (ESS)

An Execution Support System (ESS) is required to execute the software prototypes in PSDL. An important aspect of CAPS is the ability to demonstrate a functioning prototype of the system under design. It is not sufficient to conveniently and abstractly describe a system. Once a system is described, it is necessary to produce the system in executable form. The primary goal in CAPS is to automate as much as possible the production of the executable code. Next, it is desirable to simulate the behavior of systems which have hard real-time constraints. The ESS is designed to provide this facility.

The ESS consists of three interrelated parts, one of which is the subject of this paper. Figure 2 illustrates the relationship between the components of the ESS. Each element of the system and its' function will be briefly described.

The Ada implementation of such aspects of real-time systems as PERIOD, MET, MCP, MRT, and TIMER is not trivial. Ada DELAY by itself has no upper bound but is a lower bound on the delay implied. The Ada DELAY and SELECT constructs cannot be used to implement these performance constraints directly for a system of operators. The use of the type DURATION allows the approximation of an interval in a loop construct but it is not as flexible as needed. The use of TASKS in Ada provides more capability through the use of conditional entry calls. The problem with these constructs is that they require a good deal of effort on the part of the programmer to implement, and the program is operating at the mercy of the Ada run-time system. If the designer is required to invest nearly as much effort into the creation of the prototype as the development of the system itself, there is no advantage to prototyping.

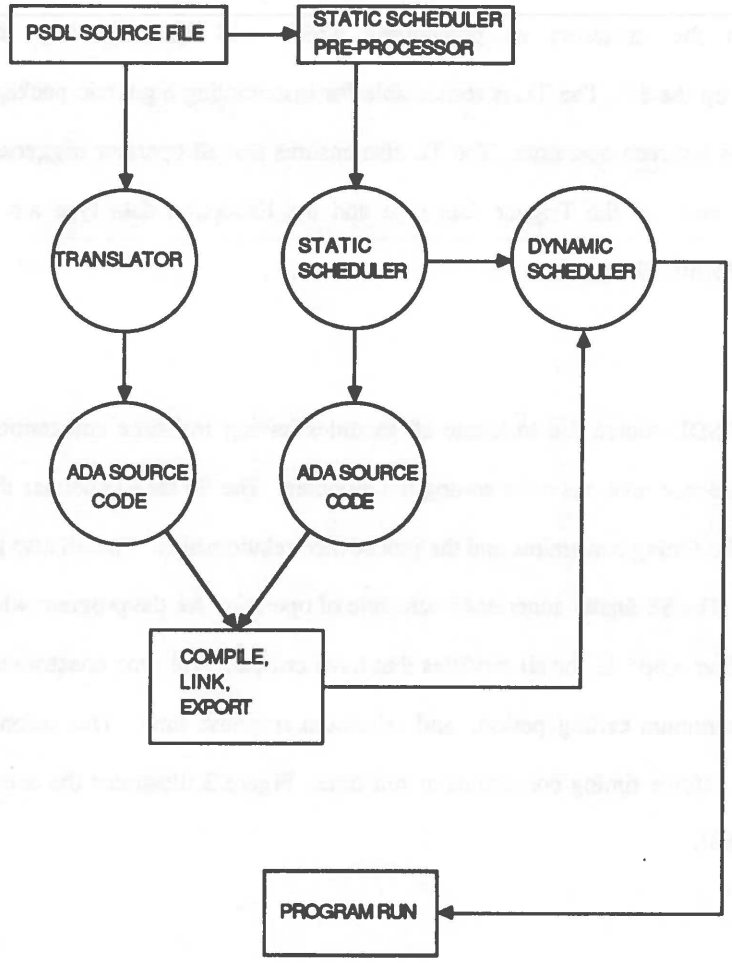

Figure 2. Execution Support System (ESS) structure

Furthermore, the Ada run time system will not guarantee that the prototype design behaves in exactly the same manner as specified. A Static Scheduler (SS) and a Dynamic Scheduler (DS) are needed to ensure that the prototype functions within the real-time constraints applied to the design. These work in conjunction with the Translator to produce executable Ada programs from PSDL specifications.

3.1. Translator

The Translator (TL) converts PSDL source code into Ada source code. Output from the TL is provided to the Ada compiler/linker along with some additional information from the Static Scheduler (SS) to

produce Ada object code. The object code is then exported to the operating system and can be run for test and demonstration purposes. The TL passes real time constraints through without translation. The TL creates code to implement the operators as procedures which will be called by the main subprogram/schedule created by the SS. The TL is responsible for instantiating a generic package which models the data stream buffers between operators. The TL also ensures that all operator triggering conditions are encoded correctly, and that the Trigger data type and the Exception data type are properly encoded for the final model[Moffitt, 1988].

3.2. Static Scheduler

The SS examines the PSDL source file to locate all modules having real-time constraints, and to determine if any special precedence relations exist among the modules. The SS then generates the necessary Ada code to implement the timing constraints and the precedence relationships. The SS also generates the main subprogram or task. The SS finally generates a schedule of operation for the program which takes into account the worst case time schedule for all modules that have critical, real-time constraints such as maximum execution time, minimum calling period, and minimum response time. This information is encoded into the modules to enforce timing constraints at run time. Figure 3 illustrates the action of the SS[Janson, 1988, O'Hern, 1988].

3.3. Dynamic Scheduler

The Dynamic Scheduler (DS) operates at runtime along with the prototype model. It is designed to control the execution of all non-critical operators within the program. A non-critical operator is one which is not subject to hard real-time constraints. The DS is invoked each time there is spare time within the static runtime schedule created by the SS. At that time DS commences execution of the next available module in its set of operators and continues to invoke non-critical modules until the available time is exhausted. At that point, operation of the DS is interrupted and control is returned to the SS to continue the time critical operations[Eaton, 1988].

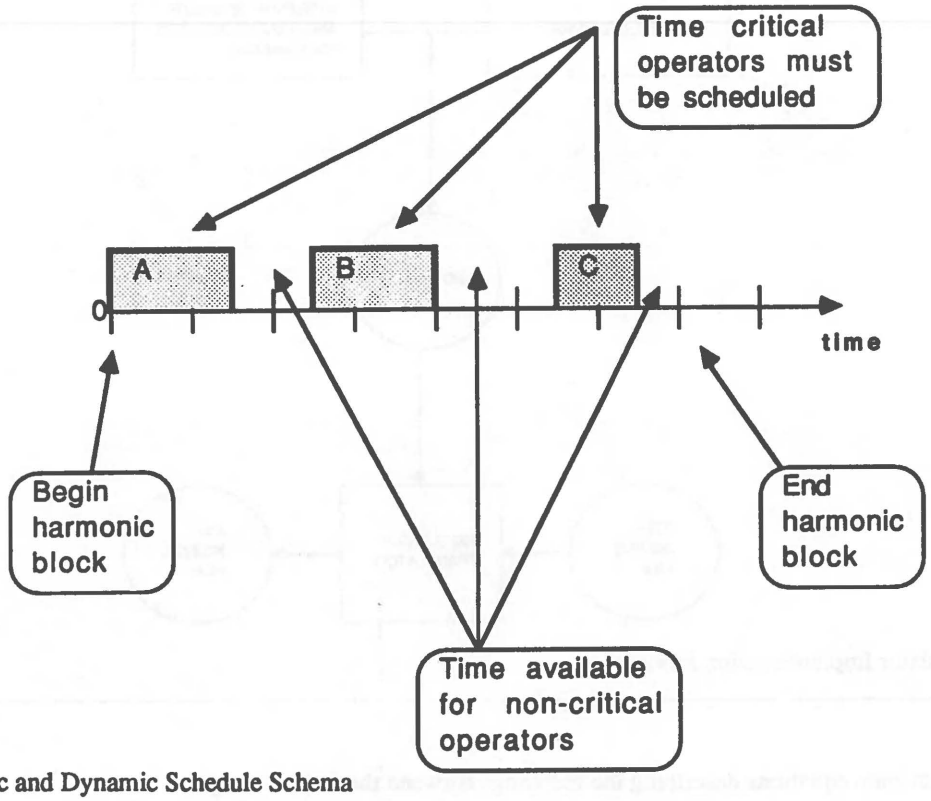

Figure 3. Static and Dynamic Schedule Schema

4. Translator Generation

The Translator (TL) is created using an automated translator generator called KODIYAK. KODIYAK was developed by Robert Herndon at the University of Minnesota as a doctoral dissertation [Herndon, 1988]. It is available as a research tool and is quite effective. The system is based on Knuth's work in attribute grammars [Knuth, 1968]. It utilizes a version of Jalili's algorithm [Jalili, 1983] to evaluate the semantic tree create the translator[Reps, 1983].

The TL implementation is straightforward. Figure 4 illustrates the process. The steps involved in producing the translator are:

(1) Describe the source language (PSDL) grammar in AG form.

(2) Create mappings between the grammar of the source language (PSDL) and the target language (Ada).

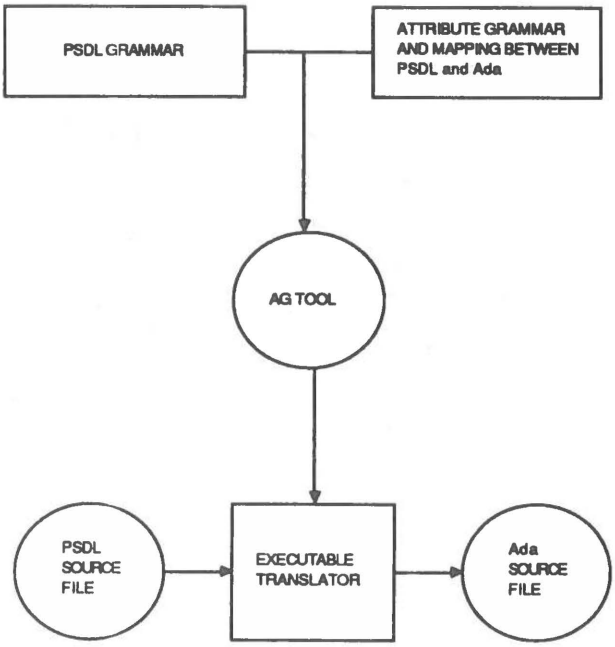

Figure 4. Translator Implementation Process

(3) Create attribute equations describing the mappings between the two languages.

(4) Combine the source language grammar description with the attribute equations into a text file which can be processed by the AG tool.

(5) Process the text file with the AG tool to produce the executable translator

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.

5. Mapping Between PSDL and Ada

It is necessary to consider how various major elements of the PSDL may best be implemented in Ada. Once a satisfactory implementation strategy is adopted it will be possible to create the attribute equations required in the Kodiyak translator generator[Herndon Berzins, 1988]. Thus, a translator can be

created which will empirically demonstrate the feasibility of the TL and the ESS.

5.1. PSDL Operator

The PSDL Operator can be implemented by producing an Ada procedure. This procedure contains code to implement any PSDL input or output conditional statements. It also contains code to check the validity and availability of data for NDF control. Before presenting an example of this construction it will be necessary to describe the implementation of the PSDL data streams.

5.2. PSDL Data Stream

There are two different type data streams in the PSDL schema. One is a FIFO queue while the other is the sampled stream. Therefore, two different generic queue models are required. One of these receives and transmits data without condition. This is the sampled stream, and will be referred to as a simple queue. Each data value in the simple queue may either be read many times or not at all. The second queue model will have a Boolean flag indicating whether or not it has been written since the last read operation or whether it has been read since the last write operation. This is the FIFO queue. It is used for Natural DataFlow Control (NDF) of operators. The Boolean flag is necessary since delivery at least once, but only once, of each data value sent through the queue is required in natural data flow. If there is a violation of the FIFO rule, then the Boolean flag will result in the queue raising an exception. There are two possible exceptions. One will be identified as Underflow, and the other as Overflow. Underflow will be raised if the consumer operator attempts to read the queue before it has been updated by the producer operator. Overflow will be raised when the producer attempts to write to the queue before the consumer has read the previous data value.

The translator must have some basis to select the appropriate queue for a given data stream. If an operator contains the TRIGGERED BY ALL tokens then FIFO queues will be selected for the streams listed following the ALL token. If the operator contains the TRIGGERED BY SOME tokens then simple queues will be selected for the data streams. A third condition is if the operator contains no TRIGGERED BY tokens. In this case simple queues will be selected. For example, in Figure 5, operator T has four input streams. The specification for T is, TRIGGERED BY ALL D,F,H. The translator will select FIFO queues for streams D,F, and H. Stream G will be a simple queue. 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 simple. Again in Figure 5, operator FF has two input streams. The specification for FF lacks a TRIGGERED BY token. Therefore, all the streams are simple streams. Thus, if the operator specification lacks the TRIGGERED BY token, or contains the SOME token, the streams will be simple. If a stream is not listed in the ALL specification it will be simple. Only when the operator contains the ALL token will a FIFO queue be selected. Note that it is the triggering conditions for the consumer operator that determine the type data stream(s) that exist between any two operators.

The data streams are modeled as a generic package containing a queue procedure in Ada. This construction is not sufficient. The SS and DS have generated a schedule for the time critical operators and this

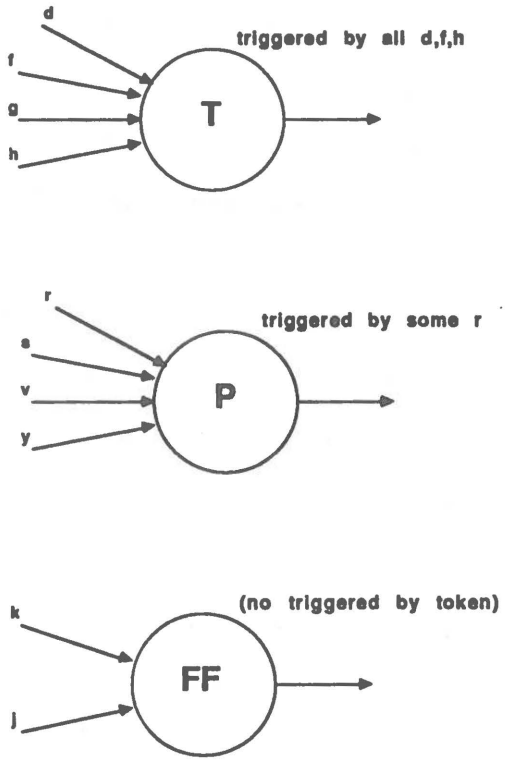

Figure 5. Queue selection based on the "TRIGGERED BY" construct

schedule is enforced to ensure real-time constraints are met. Some operators do not have time critical constraints. These operators run when none of the time critical operators are enabled. It is possible that a time critical operator is the consumer of data from a non-time critical operator. The time critical operator has priority and is scheduled to run by the SS on some repetitive cycle. The non-time critical operator is fired, as convenient for the DS, in the excess time in the main schedule. Suppose a non-time critical operator is called and is attempting to write to the data stream, when it is interrupted by the DS in order to run a time critical operator. Also suppose that the time critical operator is the consumer for the data from the non-time critical operator. When the consumer attempts to read the queue, the results will be uncertain.

This difficulty can be overcome by making the generic queue into an Ada task with a high priority. This task will be called a buffer task. The task is then enclosed as a generic package which can be generically instantiated as before. The difference is that the producer and consumer operators will use entry calls to write to or read from the buffer. 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. The operation time for any buffer task should be very short, so there should be little time penalty to the scheduled operation of the program. On the other hand, buffer operation is protected from interruption and the operators are unlikely to get uncertain results from reading them.

5.3. Buffer Selection

A problem which arises in buffer selection is illustrated in Figure 6. In this case we have the decomposition of an operator 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 includes the tokens TRIGGERED BY ALL A. Also suppose that operator BB does not contain the TRIGGERED BY ALL tokens. When the TL 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, and they have not been implemented with the same type buffer. The TL does not check inheritance rules. In operation data would be placed onto A and would then be passed to A' and into BB. The results of this translation will be uncertain. It may present no difficulty or may behave erratically. The user must prevent this type of error by ensuring that operators which result from the decomposition of higher level operators have the same triggering conditions at the input in order to prevent the buffer mismatch just

demonstrated. This difficulty only arises for lower level buffers which mirror the input buffers of the highest level operator of which they are a part. 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 the specification requirements of operators CC and DD in Figure 6. However, if A is TRIGGERED BY ALL A, then BB must be TRIGGERED BY ALL A'. It is a rule which the designer must enforce at this point. A utility similar to lint for the C language could be developed to check for this type inconsistency and incorporated into the ESS as an automatic part of the prototype translation, compilation, and export facility.

5.4. State Variable Implementation by Buffer

A final issue in data stream implementation is PSDL state variables, designated by the token, STATES INITIALLY. Each state variable will have its own buffer task. An example is seen in Figure 6.

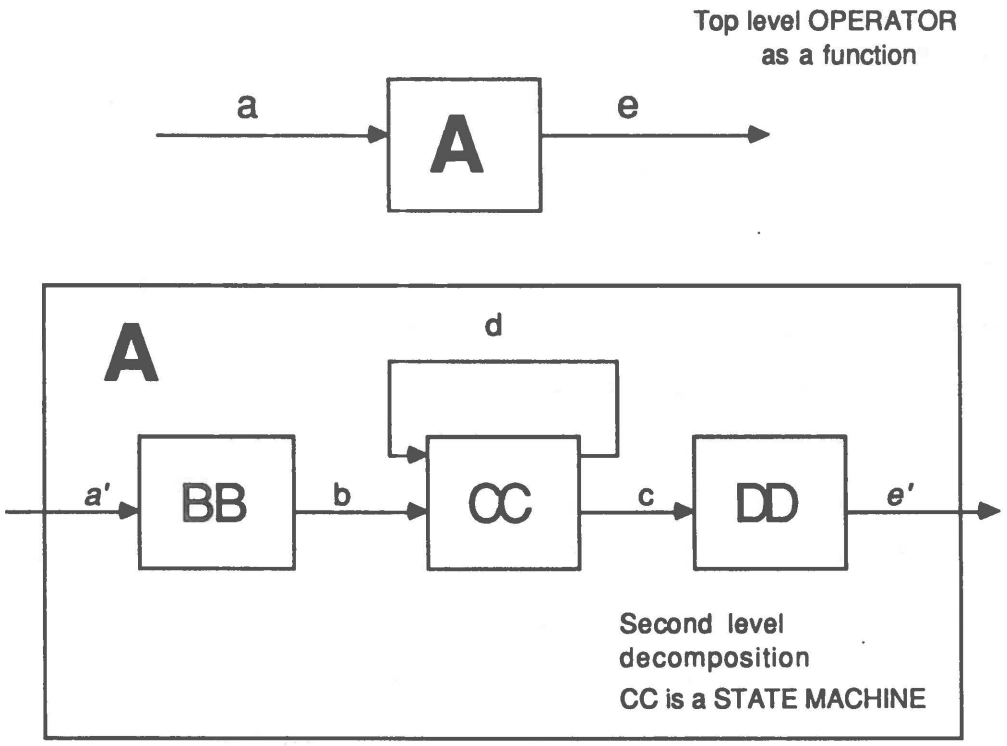

Figure 6. Buffer Selection Conflicts

Operator CC is a state machine. It has a state variable which is transmitted along buffer task D. The value of the data type 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 will be produced by the TL for all PSDL prototypes. It will contain a series of statements to put the correct initial values into the appropriate buffer tasks. If there are no state variables in the program, the procedure will simply be empty. The SS will always call PRELOAD before the execution of any schedule it creates for the prototype. The preloading procedure will not be part of the schedule proper. It will run one time only to initialize the state buffers and will not be run again unless the prototype program is restarted from the beginning.

5.5. Timers

A TIMER module must be implemented. The purpose of TIMER is to measure elapsed time between two events, the length of time an operator has been in a particular state, or to act as a conditional guard for operator firing. The four primitive operations for the timer are START, STOP, RESET, and READ. It will use the Ada standard package CALENDAR to access the system clock. The timer will have a Boolean run switch, a starting point, and a grand total.

At START, the Boolean run switch will be set to true, the system clock read and the value of the reading stored as the initial starting point. At some time later a READ is performed. The system clock will be read and the value of the initial reading subtracted from it to calculate the elapsed time. The initial value will not be changed. Elapsed time is added to the grand total and output. At STOP, the system clock is read and the elapsed time is added to grand total. The run switch is also set to false. At a subsequent START, the system clock will be read and used as the new starting point. The grand total will not be disturbed. The RESET operation will stop the timer and set the grand total to zero. TIMER is an Ada generic package, which can be instantiated wherever needed.

6. Generating Translator using the Mapping and the Translator Generator

The translator is generated with an attribute grammar based software tool[Herndon, 1985], translator generator, Kodiyak. 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 target language Ada according to the mappings derived in the previous section. The actual production of the executable file is largely accomplished by LEX and Yacc in the UNIX. Kodiyak adds the Kodiyak language which is used to describe the input language in AG form. 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, 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 assigned one attribute, trn, which is type string. Kodiyak allows attributes to be either string or integer 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 type 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.

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 Input File For the Kodiyak

7. Conclusions

Currently the CAPS is under development as a series 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 translator. However, some improvement in the translator generator is needed. The error messages returned to the user when the translator is applied to a syntactically incorrect input file cryptic at best and need improvement. The simple statement "syntax error" is not particularly helpful in debugging the input PSDL file. Efforts to integrate the various parts of the 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 prototyping environment.

The automated facility to translate a prototyping language into an underlying implementation language is feasible, and is a working reality.

BIBLIOGRAPHY

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

Luqi, Berzins, V., Yeh, R. *A Prototyping Language for Real-Time Systems,* to appear in IEEE TSE, 1988. Also Technical Report 86-04, University of Minnesota, 1986.

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

Herndon, R. *Automatic Construction of Language Translators,* Ph.D. dissertation, University of Minnesota, 1988.

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

Herndon, R., and Berzins, V. *AG: A Language Based on Attribute Grammars,* to appear in IEEE TSE, 1988.

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

Knuth, D. *Semantics of Context-Free Languages,* Math. Syst. Theory 2,2, June 1968, pp. 127-145.

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

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

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

Luqi, and Berzins, V. *Rapid Prototyping of Real-Time Systems,* to appear in IEEE Software, 1988. Also Technical Report NPS 52-87-005.

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

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

Reps, T. *Generating Language Based Environments,* (ACM doctoral dissertation award; 1983). Amherst: University of Massachusetts, 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

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

National Science Foundation
ATTN: Dr. Kent Curtis, Director of Computer
and Computation Research
1800 G Street NW
Washington, DC 20550

1