Legacy 실시간 소프트웨어의 운영체제 호출을 Ada로 번역하기 위한 방법론

이 문 근[†]

요 약

이 논문은 운영체제에 대한 호출들로 표현된 소프트웨어의 병렬성을 Ada로 번역하기 위한 방법론을 기술하고 있다. Legacy 소프트웨어들에 내재하는 병렬성은 주로 병렬 process 또는 task들을 제어하는 운영체제 호출들로 표현된다. 본 논문에서 다루고 있는 예로서는 C 프로그램내부에서 사용하는 Unix 운영체제에 대한 호출과 더불어 CMS-2 프로그램에서 사용하는 ATES나 SDEX-20 운영체제의 Executive Service Routine들에 대한 호출들을 볼 수 있다. 소프트웨어 이해를 위한 다른 연구에서는 legacy 소프트웨어에 있는 운영체제호출을 또 다른 운영체제에 대한 호출로 번역하는 데에 역점을 두고 있다. 이런 연구에서는 소프트웨어를 이해하기 위해서 소프트웨어가 수행되는 운영체제에 대한 이해가 필수적으로 요구된다. 그런데 이런 운영체제는 보통 매우 복잡하거나 체계적으로 문서화되어 있지 않다. 본 논문에서의 연구는 legacy 소프트웨어에 있는 운영체제호출을 Ada 메커니즘을 이용한 동일한 프로토콜로 번역하는 데에 역점을 두고 있다. Ada로의 번역에 있어 이러한 호출들은 메시지에 기초한 kernel 중심 구조의 scheme에 맞는 의미적으로 동일한 Ada 코드로 대표된다. 번역을 용이하게 하기 위하여 데이터 구조, task, procedure, message들을 위해 library에 있는 template들을 사용한다. 이 방법론은 소프트웨어 재·역공학측면에서 운영체제를 Ada로 modeling하는 새로운 접근방식이다. 이 방식에는 소프트웨어 이해를 위하여 기존 운영체제에 대한 지식이 필요하지 않다. 왜냐하면, legacy 소프트웨어에 내재했던 운영체제에 대한 종속성이 제거되었기 때문이다. 이렇게 번역된 Ada 소프트웨어는 여러 Ada실행환경에서 이식이 가능하고 또한 소프트웨어들간에 상호작동성이 좋다. 이 방식은 다른 legacy 소프트웨어 시스템의 운영체제호출들도 처리할 수 있다.

A Methodology for Translation of Operating System Calls in Legacy Real-time Software to Ada

Moon-kun Lee[†]

ABSTRACT

This paper describes a methodology for translation of concurrent software expressed in operating system (OS) calls to Ada. Concurrency is expressed in some legacy software by OS calls that perform concurrent process/task control. Examples considered in this paper are calls in programs in C to Unix and calls in programs in CMS-2 to the Executive Service Routines of ATES or SDEX-20. Other software re/reverse engineering research has focused on translating the OS calls in a legacy software to calls to another OS. In this approach, the understanding of software has required knowledge of the underlying OS, which is usually very complicated and informally

† 정 회 원: 전북대학교 컴퓨터과학과
논문접수: 1997년 5월 15일, 심사완료: 1997년 11월 10일

documented. The research in this paper has focused on translating the OS calls in a legacy software into the equivalent protocols using the Ada facilities. In translation to Ada, these calls are represented by Ada equivalent code that follow the scheme of a message-based kernel oriented architecture. To facilitate translation, it utilizes templates placed in library for data structures, tasks, procedures, and messages. This methodology is a new approach to modeling OS in Ada in software re/reverse engineering. There is no need of knowledge of the underlying OS for software understanding in this approach, since the dependency on the OS in the legacy software is removed. It is portable and interoperable on Ada run-time environments. This approach can handle the OS calls in different legacy software systems.

1. Introduction

This paper describes the methodology for translation of concurrent software expressed in operating system calls to Ada.

Concurrency-related OS calls generally have been informally documented. This has made understanding and analysis of such concurrent programs extremely difficult. The translation to Ada simplifies understanding of the role of concurrency. Further, the software becomes independent of OS used with the hardware.

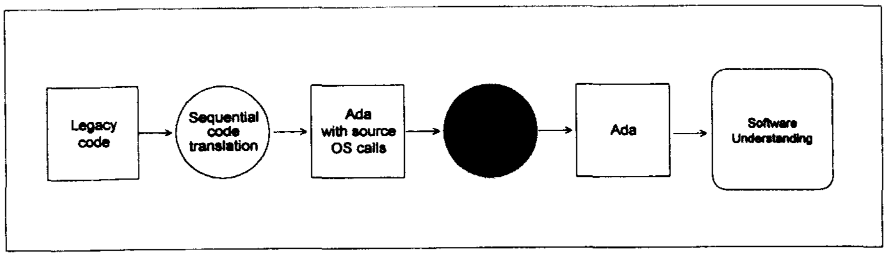
The immediate motivation for the work described in this paper has been due to the need for automatic translation of concurrent real-time U.S. Navy software in CMS-2 [Nav90a, Nav90b] into Ada [DoD83]. Embedded in the CMS-2 code are concurrency-related calls to ATES [GEA88] or SDEX-20 [Unisys, Univac] OS. Such programs are widely used in U.S. Navy mission-critical applications. The translation is needed for modernization of these systems.

Still another motivation has been due to the relatively new field of Software Reverse Reengineering, possibly, supported by Software Reverse Engineering [ChCr90]. Its objective is to process existing software automatically, or semi-automatically, in order to obtain modern software, for the same or new application for execution on a high speed distributed network.

The research reported in this paper is a part of the overall reverse engineering processes in Software Re/reverse-engineering Environment (SRE) [CCCC94, Lee95, LPL95]. SRE involves code translation and creation of a software model that is used to provide a number of capabilities: software analysis, facilitation of software understanding, documentation of the software, reorganization and restructuring of the software and interfacing with other software.

The translation of concurrency-related OS calls to Ada is shown in Figure 1. First, the sequential part of the legacy code is translated into Ada code. The OS calls in the source code remain OS calls in the Ada code. Next the concurrency, expressed in OS calls, is translated into Ada. Finally, the generated Ada code is being analyzed to facilitate software understanding.

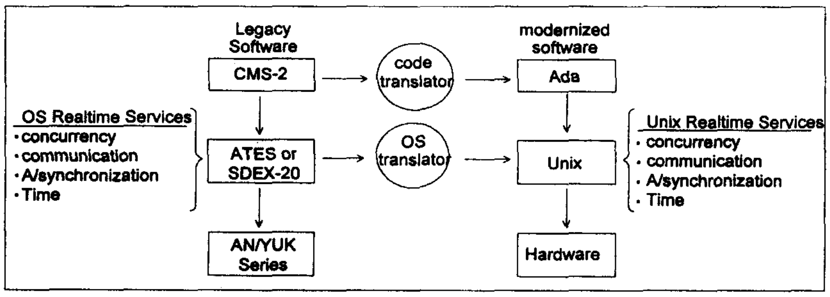
The translation of OS call is only partly feasible in some instances. To illustrate this point, the paper shows an example of the translation of the Unix *fork*



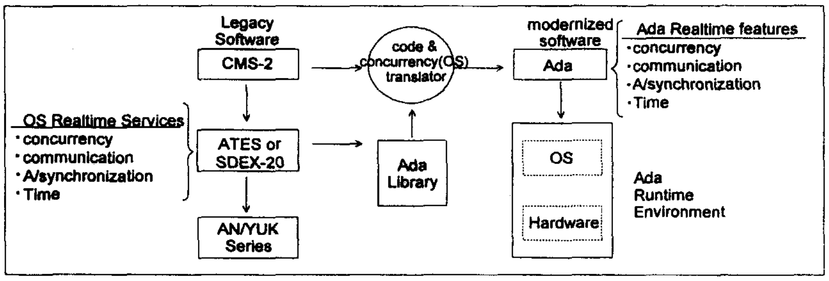
(Fig. 1) Overview

call into Ada [LMKQ89, Ste90, Sun90, KePi84]. The computational model of Unix differs greatly from the computation model of Ada. For example, Unix tasks are much larger granular objects than Ada tasks: a Unix task has its own address space and maintains by default information about open files, related tasks, and signals [LMKQ89, Ste90, Sun90, KePi84]; no such information is available by default with Ada tasks and in most implementations Ada tasks share the same address space. More fundamentally, an OS controls tasks switching and can make decisions when dispatching a task from the *Ready* state to the *Running* state (example of such decisions are to suspend or to terminate a task). No such fine grained control is possible in Ada since no facilities are provided by the language to control task switching (Ada 9X [Ada91] will correct to an extent this problem); even the drastic *Abort* statement does not take effect until the aborted task, on its own, reaches a synchronization point. One objective of the paper is to clarify and bridge over such differences.

The outline of this paper is as follows. Section 2 surveys related research on this research topic. Section 3 states a technical problem. It establishes the requirements for the translation by defining the functional equivalence of the source and target software. Section 4 presents the translation process. It consists of generating Ada tasks and procedures to implement concurrency-oriented OS calls. The translation replaces concurrency-related OS calls with respective Ada tasks, procedures and messages. It creates tasks for the source program processes and for synchronizing these processes. Section 5 describes the translation process in greater detail. It uses an Ada template library for synthesizing the concurrent aspects of the code. The Ada library contains the code needed generally for translating concurrency-related OS calls. Examples are given of procedures for translating several Unix OS calls. The methodology is applicable to other source software languages and OS (e.g. CMS-2 with ATES



(a) OS-to-OS Translation



(b) OS-to-Ada Translation

(Fig. 2) Two approaches for translating OS calls in legacy software and SDEX-20). Section 6 presents a C/Unix example for Producer-Consumer problem. The Section refers to templates for Unix concurrency-related calls. Examples of seven of these calls are shown defined by respective procedures. The implementation of procedures for all of the Unix calls and their placement in the Ada library is required for attaining the full translation of Unix into Ada. Finally, conclusion and future research are presented in Section 7.

2. Related Research

The research problem in this paper is how to translate the OS calls in legacy software to the functionally equivalent protocols in the Ada language as shown in Figures 2.a and 2.b. Legacy software utilize underlying OS services for real-time operations involved in concurrency, communication, a/synchronization, and timing. The legacy software programming languages usually do not have facilities for such real-time applications. The requests to the OS services embedded in the legacy software need to be translated to the equivalent services in other OS. The translation requires the functionally equivalent behavior of the software.

There are two approaches. The first approach is to translate the source OS calls in the legacy software to the target OS calls which a more modern software can utilize [Sam95] as shown in Figure 2.a. For example, the calls to the Executive Service Routines (ESR) of ATES or SDEX-20 in the CMS-2 [Nav90a, Nav90b] legacy software to the calls to Unix OS services in the modern Ada software. In this approach, there is no guarantee of one-to-one translation of the calls to the source OS service in the legacy software to the calls to the target OS services in the modern software. The dependency on OS in the legacy software still remains in the new software. The new software is executable only in the target OS. There is no transportability or interoperability of the software. The understanding of the software requires knowledge of the underlying OS, which is usually very complicated and informally documented.

The second approach is to translate calls to OS services in the legacy software into the equivalent protocols using the Ada facilities for concurrency, communication, a/synchronization, and timing as shown in Figure 2.b. This approach is presented in the paper. The translation replaces the OS calls with equivalent Ada code based on a message-based kernel oriented architecture. The translation utilizes code templates for data structures, tasks, procedures, and messages. Translation of each OS call requires templates of procedures that execute the respective protocol of the call. In this approach, the dependency on the OS in the legacy software is removed in the modern Ada software. The Ada software is transportable and interoperable through Ada supporting environments. The Ada software can be geographically distributed. Understanding of the modern Ada software requires only knowledge of Ada. Knowledge of OS is not required.

3. Functional Equivalence of Translation Requirements

The basic translation requirement is to produce Ada target software which is functionally equivalent to the source software. The *functional equivalence* requirements are defined as follows.

Assume first that a software specification exists that defines legal input sequences and respective outputs of the software, as well as additional timing requirements. Functional equivalence of the source and target software of the translation means that they both conform to the software specification. They both consume legal input sequences and produce respective legal outputs within timing requirements.

In practice, a software specification may not be reliable or even available. Instead, it is proposed to use the source software as its own specification. The source software is input to the translation. It is assumed to be well-tested, extensively used and highly reliable representation of the software specification, though it may be incomplete in some of the cases described below. (If the source software is modified prior to the translation, then it is required that it be tested or shown to conform to the specified computation.) We distinguish the following cases.

1) **Sequential software**: In this case the source software is a complete representation of the specification (assuming no timing requirements). The software defines all the precedences and operations needed to process legal input sequences and produce respective legal outputs. The target Ada software is then functionally equivalent to the source software as it adheres to the latter's precedences and operations.

2) **Concurrent software**: In this case there are the issue of logical correctness and time.

i) **Logical correctness**:

a) **Source software where the logical correctness is independent of the hardware speeds and OS**: In this case the source software also defines all the concurrent execution threads and the synchronizations needed for accepting legal inputs and producing legal outputs, as defined in the software specification. For given inputs the execution of the target Ada software may produce different outputs from those produced by processing the source software, but still the target Ada software is functionally equivalent to the source software as the outputs still comply with the software specification. For example, the source and target software may execute a same sequence of inputs, the former may produce sequence of different outputs than the latter, but they both conform with the software specification.

b) **Source software where the logical correctness is dependent on the hardware speeds and OS**: The source software programmer may have relied on delays, due to relative speeds of the hardware in executing portions of the software, and has omitted entering in the code the respective synchronizations that will make up a condition independent of hardware speed. In this case the source software is lacking some synchronization statements to retain the logical correctness independent of the hardware and OS used. These synchronization statements must be added in order comply always with the software specification.

ii) **Timing**: The timing requirements are documented in the software specification, but typically not in the source software. The needed capacity of the network's processors and communications must be determined for executing the Ada target software while guaranteeing the timing requirements.

To achieve functional equivalence, the translation is based first on very close adherence to the memory and manipulation in the source software. *Close adherence* to the source software means that all the entities, operations, and the precedences of the source software are represented in the graphic software model using Ada semantics. They are as follows:

1) Declarations of same named variables as declared in source software, using the same memory layout, and same scope (same shared memory).

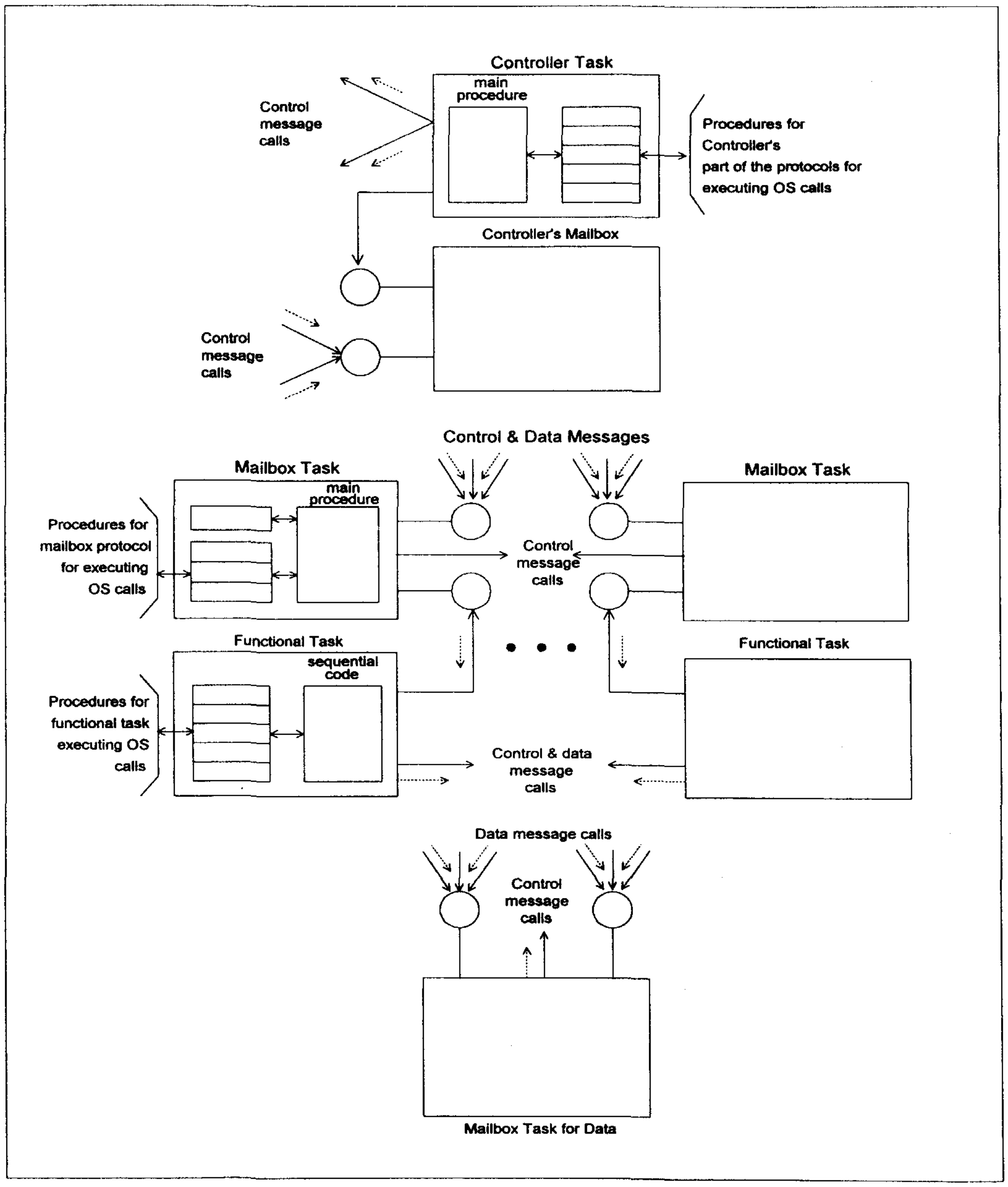
2) An Ada task and communications to perform the functions of the OS calls used in source software.

3) Ada tasks to represent each of the concurrent processes in the source software.

4) Ada procedures and functions to represent respective procedures and functions in the source software.

5) Ada I/O to represent I/O devices in the source software.

6) Ada transformation statements (executable statements) for each data or control transformation statement in the source software. They perform the same operations in Ada in the same sequential order.



(Fig. 3) Ada's tasks, procedures, and messages to implement OS calls

Next, the output of such a translation needs then to be analyzed or tested to determine whether correctness is independent of hardware speeds.

Finally the needed capacity of the hardware to meet timing requirements must be determined. These capabilities are discussed in detail in [Lee95].

4. Strategy for Translating Concurrency-Related OS Calls to Ada

This section defines the Ada entities that are synthesized to accomplish the protocols of OS calls.

The translation replaces processes in the source software with Ada tasks call *functional tasks.* Additional tasks are created for a *controller* to execute OS calls and for buffering of communication between other tasks. Procedures are added for each OS call. They are illustrated in Figure 3 and discussed below. Large rectangles in the figure denote Ada tasks; circles denote task entries; communication paths denote flow of messages (showing direction of the call and direction of the message); columns of rectangles (inside tasks) denote procedures for executing OS calls.

The source software OS calls are performed by generating in each processor an Ada task called *controller.* It is shown at the top of the Figure 3. It only performs the concurrency related OS calls that are actually used in the source software. (OS error processing and I/O are not considered.) The controller task plays the role traditionally played by the kernel in OS. That is, it provides the basic mechanisms for supporting task creation, interaction, and termination, and for communication with the program's environment.

The OS also performs dispatching of processes and uses a variety of underlying systems. However, these services are provided by the Ada compiler when generating object code by inserting into the code OS calls tailored for each vendor's OS, hardware and communications. These capabilities therefore are not included in the translation. The scheduling by the source OS may differ from that of the target Ada Program but should not affect the correctness of the software, although it may affect the timing. If the source OS supports priority assignments for processes, then corresponding priorities are also generated for respective Ada tasks. If the source language or OS supports priorities for messages, then they are included in the operations of the mailbox task, as described below:

1) The **messages** exchanged between tasks are of two kinds:

i) ***Control messages*** for interpreting OS calls; these messages are exchanged between the controller tasks and the functional tasks.

ii) ***Data messages*** for communicating variables among processes as specified in the source program; these messages are sent or received by the functional tasks that represent source software processes.

2) The **controller task** contains in its body a main loop that:

i) Receives a control message from other tasks (via the controller's mailbox task) to perform the equivalent of an OS call.

ii) Calls a procedure that executes the protocol of the OS call. The protocol may involve sending control messages to tasks and receiving acknowledgements of protocol steps.

Thus, the Controller task is implemented as a nonolithic kernel in the sense that it can execute the protocol of a single OS call at a time. Most of the work of the call is done outside the Controller which only routes messages and updates task control information. If it will prove necessary, the design of the Controller will be reconsidered to become an interacting family of tasks, each with a different priority, corresponding to the priority of the callers and to specific segments in the execution of the call. The declaration of a controller task is inserted at the beginning of the target Ada concurrent software. The controller is dynamically provided with data on each task being created. This data is similar to a Process Control Block of an OS [LMKQ89, Ste90, Sun90]. It is called *Task Control Block* (TCB).

Processes created by OS calls in the source program are translated into declaration of respective Ada tasks, the *functional tasks.* They are illustrated in the middle of Figure 3. The declarations of functional tasks are inserted in the places where there are calls to the OS in the source software to create the respective processes. Each functional task contains a main procedure that corresponds to the sequential execution code of the respective process in the source program. In Ada, the sequential execution code is contained in the body of the functional task. The functional task body may contain calls to procedures that send control messages to the controller task, thus causing execution of the protocol of the respective OS call. A functional task may be assigned a priority, as indicated, by a respective source software OS call. The inserted declarations of functional tasks create the tasks dynamically and their creation is reported to the controller task.

An originating task may call the controller task to execute an OS call on a destination task. Each functional task must check periodically if it has a waiting command from the controller. If one exists, then the functional task must execute the command. An example of such a command is a call in one functional task for suspending or terminating another functional task. When the destination functional task receives the command, it suspends or terminates itself normally. The checks for existence of a waiting message add overhead to the execution of these commands. The required response time to these commands determines the frequency of checking for such messages. It must be compensated for by using much faster hardware with target Ada software than the hardware used with the source hardware.

For the reasons below, it was determined necessary to have all communications between functional tasks or between the controller and a functional task to go via a *mail box* task. The mailbox task contains intelligence to handle the following:

1) Recognizing and buffering variable length data messages.

2) Recognizing and buffering control messages and giving top priority in delivery of control messages.

3) Delivering data messages in a priority order, in accordance with the message priority requirements of the source software OS. Same priority data messages are delivered in first-in first-out order.

4) Acknowledging receipt or delivery of message corresponding to requirements of the source software OS calls. Note that this can support both guaranteed delivery of messages as well as serving blocked or unblocked communication commands.

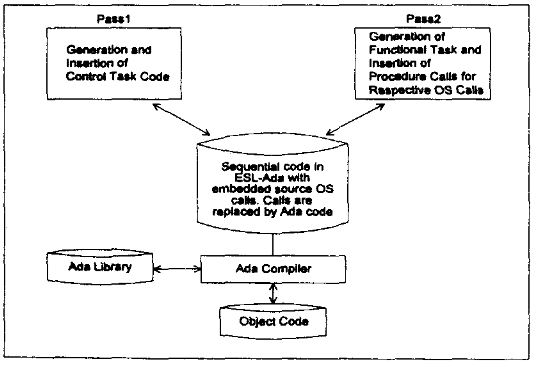
5) Receiving, interpreting, and acknowledging control messages directed to the mailbox task itself. This is necessary to suspend, continue or terminate a mailbox.

Mailbox tasks are created dynamically. They are reported to the controller for each control and functional task. A mailbox task is created at the initialization of the respective controller or functional task. Thus, there is one mailbox task for the controller task and one for the functional task. An additional mailbox task may be created to establish a sending and receiving communication path between multiple processes in the source software. This task is also created dynamically in the corresponding place where the source software communication path is declared in the source software. The mailbox tasks are also shown in Figure 3. Each mailbox task has an entry point for calls of incoming messages and an entry point for delivery of messages.

The use of mailbox may at times add an overhead of as much as double the communication time between tasks. This overhead must also be compensated for by use of hardware that is considerably much faster than the hardware used by the source software.

The selection of Ada primitives and their method of synthesis takes into account minimizing the overhead in execution, and maintaining, or even improving the understandability of graphic model of the software. We distinguish between two types of primitives: those that are generic to many OS's, and those that are specific to a selected OS. There are some structures and operations that are common to a number of OS's. The translation of each OS call requires composing its own procedures that implement the respective protocol of the call. Both the generic code and the OS call procedures are placed in the Ada library as discussed in Section 5. The procedures that execute OS calls are shown in Figure 3, inside the respective boxes.

An OS call may not be fully translatable to Ada, or translatable only in a restricted way. For example, Section 6 refers to the Unix *fork* call which involves creating a new process. It consists of copying an executable image, creating an appropriate task control structure, and rescheduling. This cannot be done in general directly in Ada. But this can be done in Ada if the code executed in the child thread consists of a call to a predefined pure procedure, i.e. all data used in the procedure are either local or an explicit parameter of the procedure (*Out*, and *In Out* parameters of the procedure must have been copied before the procedure in called.).


Pass1
Generation and Insertion of Control Task Code
Pass2
Generation of Functional Task and Insertion of Procedure Calls for Respective OS Calls
Sequential code in ESL-Ada with embedded source OS calls. Calls are replaced by Ada code
Ada Library
Ada Compiler
Object Code

(Fig. 4) Conversion of OS calls to Ada

5. Implementation of Translation of Concurrency-Related OS Calls

5.1 Two Pass Process

The translation of OS calls is shown in Figure 4 as a circle titled *Concurrency Translation*. The output of this process is the *Elementary Statement Language* for Ada (ESL-Ada) [CCCC92, Lee95] of the software obtained from translating the source sequential code into Ada [CCCC94, Lee95]. Note that ESL-Ada a graphical language to represent Ada software based on *Entity-Relation-Attribute* (ERA) graphs [Che76], where the statements are represented as nodes, and the relations between statements as edges, called *tuple*. The output includes source OS calls embedded in the sequential Ada code in the graphic software model. This concurrency translation process replaces the source OS calls in the ESL-Ada with Ada code.

The translator process is illustrated in further detail in Figure 4. As shown, the ESL-Ada software model is updated in two passes. The first pass consists of:

1) Scanning the ESL-Ada graphic model of the software to find:

i) the concurrency-related OS calls that are used in the source software,

ii) which calls create processes and therefore must be replaced by functional task declarations, and

iii) the beginning and end of the sequential thread of execution code performed in each of the source processes.

This information is tabulated for use in Pass 2.

2) Generating the controller task which contains calls to the procedures that interpret all the OS calls that are used in the source software. As noted above the code in the library is for a single processor. Only a controller task exists for each processor in the network. It is generated in place and not included in the Ada library. The controller task declaration is generated in Pass 1. Its specification is inserted at the beginning of the ESL-Ada model of the software. It contains procedures for only the OS calls in source software.

3) Creating a mailbox task for the controller task.

The second pass consists of:

I. Data declarations for use in:

- Mailbox tasks, for receiving, sending and buffering data & control messages
- Generic package of functional task
- Controller task

II. Type of precedure for:

- Interpreting the protocols of the source Operating System calls in the:
  - Controller task
  - Functional Tasks
  - Mailbox tasks

III. Types of tasks and generic package for:

- Mailbox task
- Generic Package that contains a functional task

(Fig. 5) Summary of the Ada library

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                                  PRODUCER                                                 */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
void  PRODUCER(BUF)                            /* * * * * * * * * * * * * * * * * * * * * * */
int BUF;                                       /*                                          */
{                                              /*    WRITE PIPE                            */
        char MSG[1];                           /*                                          */
        int   FLAG = 1;                        /*    Message to be written on PIPE         */
                                               /*    Flag for while iteration              */
        while  (FLAG)                          /*                                          */
        {                                      /*    Repeat until false                    */
          get (MSG);                           /*                                          */
          write (BUF , MSG , sizeof (MSG) );   /*          Input from standard input       */
          /* reset   FLAG */                   /*          write Msg on PIPE               */
        }                                      /*          Reset flag                      */
        exit() ;                               /*                                          */
                                               /*    Exit from execution                   */
}                                              /* * * * * * * * * * * * * * * * * * * * * * */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                                  CONSUMER                                                 */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
  void  CONSUMER(BUF)                          /* * * * * * * * * * * * * * * * * * * * * * */
  int BUF;                                     /*                                          */
  {                                            /*    READ  PIPE                            */
        char MSG[1];                           /*                                          */
        int    FLAG = 1;                       /*    Message to be read from PIPE          */
                                               /*    Flag for while iteration              */
        while  (FLAG)                          /*                                          */
        {                                      /*    Repeat until false                    */
          read (BUF , MSG , sizeof (MSG) );    /*                                          */
          put (MSG);                           /*          read Msg on PIPE                */
          /* reset   FLAG */                   /*          output from standard output     */
        }                                      /*          Reset flag                      */
        exit() ;                               /*                                          */
  }                                            /*    Exit from execution                   */
                                               /* * * * * * * * * * * * * * * * * * * * * * */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                                    MAIN                                                   */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
 main()                                        /*    MAIN FUNCTION                         */
        int (BUF) ;                            /*    READ/WRITE pointers to PIPE           */
                                               /*                                          */
        pipe (BUF) ;                           /*          Open a PIPE                     */
        if (!fork())                           /*          Create a child process          */
        {                                      /*                                          */
          CONSUMER (BUF[0]) ;                  /*            Calling CONSUMER function     */
        }                                      /*                                          */
        if (!fork())                           /*          Create a child process          */
        {                                      /*                                          */
          PRODUCER (BUF[1]) ;                  /*            Calling PRODUCER function     */
        }                                      /*                                          */
        wait() ;                               /*          Wait  for  a child terminated   */
        wait() ;                               /*          Wait for  a child terminated    */
        close (BUF[0]) ;                       /*          Close READ PIPE                 */
        close (BUF[1]) ;                       /*          Close WRITE PIPE                */
 }                                             /* * * * * * * * * * * * * * * * * * * * * * */
```

(Fig. 6) Source C/Unix code for producer-consumer example

1) Inserting the instantiation of respective functional tasks and their mailbox tasks in place of each OS call that creates a process. The inserted code sends a control message to the controller task, reporting the created task control block.

2) Inserting an instantiation of a mailbox task for each OS call that establishes an inter-task data messages channel.

3) Inserting in-place calls to procedures that execute every other type of OS call.

5.2 Use of Ada Library

As shown in Figure 4, the creation of Ada tasks and procedures is simplified by instantiation of predefined objects in the Ada library.

The contents of the Ada library is presented in [Lee95] but omitted here due to its extensive size. However, it is outlined in Figure 5. It contains the data declarations used in the tasks shown in Figure 4. Next, the library contains procedures for the OS calls. Finally, the Ada library contains a declaration of a generic package for a functional task and a mailbox type. Since there may be a large number of functional tasks, the use of the generic package for the functional tasks facilitates the software understandability. The instantiation of each functional task in Pass 2 requires providing the generic package with parameters:

1) a unique name for the task,

2) the name of the procedure that corresponds to the code executed in the respective source process, and

3) the names of the procedures for interpreting the OS calls by the respective functional task.

6. Example of Translating Producer-Consumer Code in C/Unix to Ada

This section illustrates the process of translating concurrent software into Ada with a C/Unix example for Producer-Consumer problem. It includes only the features related to the example. Others are listed in the Ada library in [Lee95].

6.1 Producer-Consumer Example in C/Unix

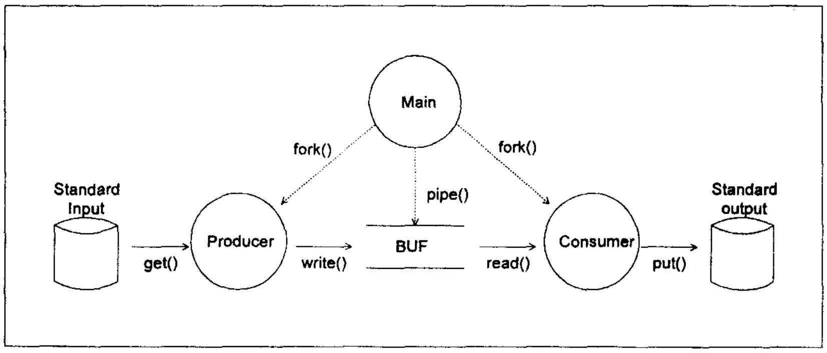
Figure 6 shows the code for a C/Unix program example for the Producer-Consumer problem. This program consists of three entities: *main*, *producer*, and *consumer*. At runtime, there are three processes running concurrently as follows:

1) A process executing the original *main*() program that creates a PIPE, and two child processes.

2) A process executing a copy of the *main*() program in which a *producer*() function is being called.

3) A process executing a copy of the *main*() program in which a *consumer*() function is being called.

These concurrent entities are called MAIN, PRODUCER, and CONSUMER processes respectively.



(Fig. 7) Overview of producer-consumer example in C/Unix

The runtime structure of these entities and their relationships are shown in Figure 7. It includes *pipe*, *fork*, and *read/write* operations.

In the following, each function and its respective operations are described:

1) **Main()**: It creates a PIPE and two concurrent processes, called PRODUCER and COMSUMER, which execute *producer*() and *consumer*() functions. It passes a WRITE pointer to the PIPE to PRODUCER and a READ pointer to CONSUMER for communication of message between PRODUCER and CONSUMER. Once both processes are active, it waits for termination of both processes. After termination, it closes the PIPE and terminates its own execution.

2) **Producer()**: It is executed in a copy of the *main*() function. In execution, a pointer to a PIPE is passed as a parameter to write messages. It writes on the PIPE a message which has been gotten from the input, in each iteration of a loop. This loop continues until *flag* condition is not met.

3) **Consumer()**: It is executed in a copy of *main*() function. In execution, a pointer to a PIPE is passed as a parameter to read messages. It reads from the PIPE a message which is to be written to the output, in each iteration of a loop. This loop continues until *flag* condition is not met.

4) **Pipe()**: It is an inter-process communication mechanism used in UNIX OS. This is one-way communication mechanism, with two pointers for READ and WRITE. It is created by MAIN, and is passed to PRODUCER and CONSUMER for communication between them.

6.2 Producer-Consumer: PASS 1

The input to Pass 1 consists of only sequential code translated to Ada. All OS calls remains commented as in same software. These OS calls are used in Pass 2 for translation to Ada.

The input of Pass 1 of the translation of the C/Unix Producer-Consumer example to Ada is shown in Figure 6. C code is translated into Ada procedures. This sequential translation is performed by a separate sequential code translator [CCCC94, Lee95]. The output of Pass 1 is organized as follows:

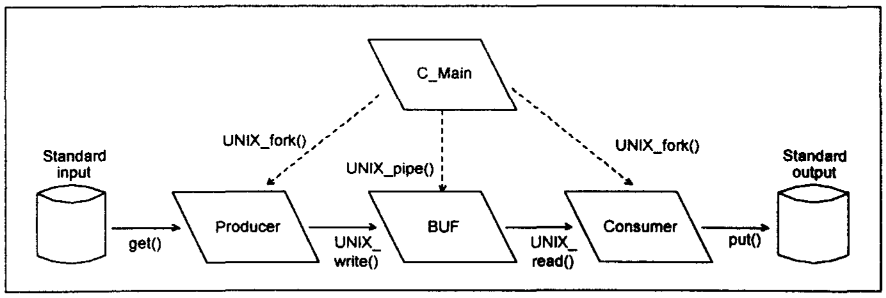
1) **A list of OS calls**: These are used to generate the body of a controller to interpret OS calls of the target Ada code.

2) **A list of procedure bodies of functional tasks**: These are actual bodies of the functional tasks in the target Ada code.

3) **The type of arguments for procedures in 2)**: These parameters are to be passed to the functional task at the time of instantiation.

Based on these information, the following tasks and procedures are generated in Pass 1:

1) **Controller**: It is a task that translates the source



(Fig. 8) Overview of producer-consumer example in Ada software OS calls. It includes only servicing concurrency related OS calls that are actually used in the source software. It performs the following operation in iteration:

i) Receives a control message from a respective task to perform an OS call.
ii) Calls a procedure that interprets the protocol of the OS call. The protocol may involve sending control message to tasks and receiving an acknowledgement of the protocol.

2) **Consumer**: It is a procedure corresponding to *consumer()* function in source software. The sequential part has been separately translated into corresponding sequential Ada code. All OS calls remain to be translated into the concurrent Ada code in Pass 2.

3) **Producer**: It is a procedure corresponding to *pro ducer()* function in source software. The sequential part has been separately translated into corresponding sequential Ada code. All OS calls remain to be translated into the concurrent Ada code in Pass 2.

4) **Main**: It is a procedure corresponding to *main()* function in source software. The sequential part has been translated into corresponding sequential Ada code. All OS calls remain to be translated into the concurrent Ada code in Pass 2.

The source software is translated into a procedure. One of the reason for this is to modularize each respective source software. This increases readability and understandability of the generated Ada target code.

The structure of the translated Ada code is shown in Figure 8. It consists of tasks: a controller tasks with a mail box, functional tasks with mail boxes, and a pipe. Note that the reason for MAIN to be instantiated as a task is due to UNIX semantics that MAIN is a process. The outputs of Pass 1 are specifications and bodies of Controller task, Producer, Consumer, and Main procedures. The outputs also include data structure declarations and type conversion functions used in Controller task and other three procedures. All remaining OS calls are translated in Pass 2.

6.3 Producer-Consumer: PASS 2

In Pass 2, all OS calls are translated according to their protocols. This translation consists of:

1) Creating a functional task and its mail box in place of each OS call to create a respective process. The functional task calls its own procedure.

2) For every call in the functional task's own procedure, insert in-place a call to a procedure that interprets the part of the OS call protocol performed by the functional task.

3) Creating a mailbox task for each multiple input/output communication indicated by an OS call.

In the following, procedures to handle Unix OS calls in Functional Tasks are listed. These are pre-defined in the Library in Appendix A of [Lee95].

1) **FORK OS call**: The *fork()* OS call is translated into a call to a function in *PRODUCER_PACK* package, which is an instantiation of package called *FUNCTIONAL_TASK_PACK*. This package is defined in the library. This instantiation requires the body of PRODUCER or CONSUMER procedure to be passed as a parameter. A call to a *Generate_Functional_Task* function in this instantiated package requires a set of proper parameters to be passed. These parameters are used to generate a desired functional task. These are i) name of child task being created, ii) a pointer to this task creating a new child task, iii) a pointer to the CONTROLLER mail box, and finally iv) a list of parameters for this PRODUCER or CONSUMER procedure in a stream of bytes. In instantiation, this newly created child task generates its own mail box, informs a controller about its own task and mail box, and finally calls the body procedure of its own. When Controller receives messages about these new tasks, calls *NEW_TCB_Rou-*

tine procedure, in library, to create a new TCB for this functional task and mail box. This TCB is used to inform parent task about its child task.

2) **EXIT OS call**: The *exit*() OS call is translated into a call to *UNIX_Exit* procedure defined in the library. This procedure generates a control message for this OS call and passes it to Controller. When Controller receives this message, it calls a procedure *SYS_EXIT_Routine* defined in the library. This procedure will perform the protocol of *exit*() OS call in

```
with LIBRARY_PACK; use LIBRARY_PACK;
with TEXT_IO; use TEXT_IO;
with Unchecked_Conversion;

procedure ADA_MAIN is
    --<other Ada code for controller and
    -- Mail Box Tasks>
    ------------------------------------------------
    -   PRODUCER
    ------------------------------------------------
    producer Producer
        (CtrlMBoxPtr  : in MAIL_BOX_P;
        FTMBoxPtr     : in MAIL_BOX_P;
        ArgList       : in MESSAGE_TAIL_T)
    is
        -- Local Variables;
        MSG  : CHARACTER;
        FLAG : INTEGER := 1;
        -- Conversion Dependent Local Variable:
        LocalArgList : PRODUCER_ARG_P
         := BYTE_Str_T_2_Producer_Arg_t(ArgList);
        ByteStream : MESSAGE_TAIL_T;
        N          : INTEGER := MSG'SIZE;
        RValue     : INTEGER ;
    begin
        while (FLAG = 1) loop
            GET(MSG);
            ByteStream := MSG_2_Byte_Str_T(MSG);
            UNIX_Write(FTMBoxPtr, LocalArgList.BUF,
                          ByteStream,N,RValue);
        end loop;
        UNIX_EXIT(CtrlMBoxPtr,FTMBoxPtr);
    end Producer;
    ------------------------------------------------
    -                  CONSUMER
    ------------------------------------------------
    procedure Consumer
        (CtrlMBoxPtr  :  in MAIL_BOX_P;
        FTMBoxPtr     :  in MAIL_BOX_P;
        ArgList       :  in MESSAGE_TAIL_T)
    is
        -- Local Variables:
        MSG  :  CHARACTER;
        FLAG :  INTEGER :=1;
        -- Conversion Dependent Local Variable:
        LocalArgList : CONSUMER_ARG_P
           := Byte_Str_T_2_Consumer_Arg_T(ArgList) ;
        ByteStream    : MESSAGE_TAIL_T;
        N             : INTEGER := MSG'SIZE
        RValue        : INTEGER;
    begin
        while (FLAG = 1) loop
            UNIX_Read (FTMBoxPtr, LocalArgList, BUF,
                          ByteStream,N,RValue);
            ByteStream := Byte_Stre_T_2_MSG(ByteStream);
            put(MSG);
        end loop;
        UNIX_EXIT(CtrlBoxPtr, FTMBoxPtr);
    end Consumer;
```

```
    ------------------------------------------------
    -                  C_MAIN
    ------------------------------------------------
    procedure C_Main
        (CtrlMBoxPtr    : in  MAIL_BOX_P;
        FTMBoxPtr       : in MAIL_BOX_P;
        ArgList         : in MESSAGE_TAIL_T)
    is
        -- Local Varibles:
        BUF     : MAIL_BOX_P;

        -- Conversion Dependent Local Variables:
        ProducerArg     :  PRODUCER_ARG_T;
        ConsumerArg     :  CONSUMER_ARG_T;
        ByteStream      :  MESSAGE_TAIL_T;
        FTaskPtr        :  STANDARD_TASK_P;
        RValue          :  INTEGER;
        Name            :  NAME_T;

        Consumer_P : STANDARD_TASK_P;
        Producer_P  : STANDARD_TASK_P;

        -- Instantiation of PRDUCER package
        package PRODUCER_PACK is new FUNCTIONAL_TASK_PACK
              (Task_Body_Procedure=>Producer);

        -- Instantiation of CONSUMER package
        package CONSUMER_PACK is new FUNCTIONAL_TASK_PACK
              (Task_Body_Procedure=> Consumer);
    begin
        UNIX_PIPE ( CtrlMBoxPtr, FTMBoxPtr, BUF, RValue ) ;
        ProducerArg.BUF        :=  BUF;
        ByteStream             :=  Producer_Arg_T_2_Byte_Str_T
                                        (ProducerArg);
        Name                   :=  Get_Task_Name;
        Producer_P:=PRODUCER_PACK.Generate_Functional_Task
                  ( Name,  FTMBoxPtr, CtrlMBoxPtr, ByteStreamP ) ;
        ConsumerArg.BUF        := BUF;
        Bytestream             :=  Consumer_Arg_T_2_Byte_Str_T
                                        (ConsumerArg);
        Consumer_P:=CONSUMER_PACK.Generate_Functional_Task
                  (Name, FTMBoxPtr,CtrlMBoxPtr,ByteStream);
        UNIX_Wait ( CtrlMBoxPtr, FTMBoxPtr, NULL, RValue ) ;
        UNIX_Wait ( CtrlMBoxPtr, FTMBoxPtr, NULL, RValue ) ;
        UNIX_Close ( CtrlMBoxPtr, FTMBoxPtr, BUF, WRITE_PIPE, RValue );
        UNIX_Close ( CtrlMBoxPtr, FTMBoxPtr, BUF, READ_PIPE, RValue );
    end C_MAIN;

    -- Instantiation of C_MAIN package:
    package MAIN_PACK is new  FUNCTIONAL_TASK_PACK
          (Task_Body_Procedure=>Main);

begin
    declare
        Name          : NAME_T;
        ByteStream    :  MESSAGE_TAIL_T;
        MainSrg       :  MAIN_ARG_T;
        C_MAIN_P      : STANDARD_TASK_P;
    begin
        Name          :=   Get_Task_Name;
        ByteStream    :=  Main_Arg_T_2_Byte_Str_T(MainArg) ;
        C_Main_p      :=   C_MAIN_PACK.Generate_Functional_Task
                  (ControllerMailBoxPtr, ControllerMailBoxPtr,
                   ByteStream) ;
    end;
    NULL;
end ADA_MAIN;
```

(Fig. 9) Target Ada code for producer-consumer example the source software OS.

3) **WAIT OS call**: The *wait*() OS call is translated into a call to *UNIX_Wait* procedure defined in the library. This procedure generates a control message for this OS call and passes it to Controller. When Controller receives this message, it calls a procedure *SYS_WAIT_Routine* defined in the library. This procedure will perform the protocol of *wait*() OS call in the source software OS.

4) **PIPE OS call**: The *pipe*() OS call is translated into an instantiation of mail box. In instantiation, it makes a call to *UNIX_Pipe* procedure defined in the library to inform Controller about this new mail box task named PIPE. This procedure generates a control message for new task and passes it to Controller. When Controller receives this message, it calls a procedure *SYS_PIPE_Routine* defined in the library. This procedure creates a new TCB for this mail box.

5) **CLOSE OS call**: The *close*() OS call is translated into a call to *UNIX_Close* procedure defined in the library. This procedure generates a control message for this OS call and passes it to Controller. When Controller receives this message, it calls a procedure *SYS_CLOSE_Routine* defined in the library. This procedure will perform the protocol of *close*() OS call in the source software OS.

6) **READ OS call**: The *read*() OS call is translated into a call to a procedure UNIX_Read defined in the library. This procedure gets a data message from a PIPE mail box by calling a READ entry. In reading, it checks for the right message by the size and type of message.

7) **WRITE OS call**: The *write*() OS call is translated into a call to a procedure *UNIX_Write* defined in the library. This procedure generates a data message to a PIPE mail box by calling a WRITE entry. In writing, it waits for an acknowledgement from the PIPE based on blocking information. In return, it receives a number of bytes written on the PIPE.

Bodies of three functional task procedures are presented in Figure 9. These are C_Main, Producer, and Consumer procedures. Only the translation of OS calls in these bodies are high-lighted with bold face. The details of procedures used are defined in the library in [Lee95].

7. Conclusion and Future Research

This paper described the methodology for translation of concurrent software expressed in OS calls to Ada, as a part of a large re/reverse engineering processes in SRE. Concurrency in some legacy software is expressed by OS calls that perform concurrent process/task control. For example, this paper presented calls in C programs to Unix. This approach is also applicable to other cases, such as calls in CMS-2 programs to the ESR of ATES. The methodology focused on translating the OS calls in a legacy software into the equivalent protocols using the Ada facilities. In translation to Ada, these calls are represented by Ada equivalent code that follow the scheme of a message-based kernel oriented architecture. To facilitate translation, it utilizes templates placed in library for data structures, tasks, procedures, and messages. This methodology is a new approach to modeling OS in Ada in software re/reverse engineering. Compared to other approaches, there is no need of knowledge of the underlying OS for software understanding in this approach, since the dependency on the OS in the legacy software is removed. It is portable and interoperable on Ada run-time environments. This approach can handle OS calls in different legacy software systems.

The research reported in this paper has a number of limitations since it focused on the translation of concurrent source software into Ada for a single processor. The future research should handle the problems of checking and modifying the software to attain independence of the logic on the source hardware and distribution of the calculations as follows:

1) **Distributing concurrent software of large scale ap-**

plications: Large-scale concurrent software is envisaged as consisting of large number of Ada tasks divided among software units. Frequently communicating pairs of tasks should be executed in co-located processors. Less frequently communicating tasks may be distributed geographically.

2) **Modifying the Ada software for independence of the hardware**: The programmer of the source software may have omitted some synchronization statements based on knowledge of the speed of the processors and the scheduling discipline of the OS used. Such omitted synchronization commands must then be added to the software in order to attain logic-wise independence of the speed of the processors used.

3) **Determining required processor and communication capacity for satisfying real-time deadlines**: The timing requirements are typically not specified in the source software. They must be obtained by the human user of the software reengineering facility from the software specification.

Acknowledgement

This research is supported in part by a research grant from the Chonbuk National University in Korea and Computer Command and Control Company in USA. I thank Professors Prywes and Insup Lee at the University of Pennsylvania and Professor Ingargiola at Temple University in USA for valuable advices and discussions.

References

[Ada91] Ada 9X Project Office. *Ada 9X Requirements Rationale*. Ada 9X Project Office, Carnegie Mellon University, Pittsburgh, Pennsylvania. May 1991.

[ChCr90] Elliot J. Chikofsky and James H. Cross II. Reverse Engineering and Design Recovery: A Taxnomy. *IEEE Software*. 7(1). January 1990. pp. 13-17.

[CCCC92] Computer Command and Control Company. *Software Intensive System Reverse Engineering*. Final Report, Contract No. N60921-90-C-0298. April 1992. Project memoranda: *Memo 2-Elementary Statement Language Internal Representation*. June 29, 1992, Memo 3-CMS-2 to ESL Translation. January 30, 1992.

[CCCC94] Computer Command and Control Company. Technical Report, *Software Reverse Engineering (SRE) Version 5.0: Demonstration Guide*. Naval Surface Warfare Center, Contract N60921-92-C-0196. May 1994.

[Che76] P. Chen. The Entity-Relationship Model: Toward A Unified View of Data. *ACM Transactions on Database Systems*. May 1976.

[DoD83] DoD. *Ada Programming Language*. ANSI/MIL-STD-1815A. January 1983.

[GEA88] GE Aerospace. *AEGIS Tactical Executive System (ATES/43) User's Mannual, Volume 1-4, System Design Guide*. GE Aerospace, RCA Electronic Systems Department, Government Electronic Systems Division, Moorestown, NJ 08057. July 1988.

[KePi84] B. Kernighan and R. Pike. *The UNIX Programming Environment*. Prentice-Hall. 1984.

[KeRi78] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice Hall. 1978.

[Lee95] Moon Lee. *An Environment for Understanding of Real-time Systems*. Ph.D. Thesis, The University of Pennsylvania. 1995.

[LMKQ89] S. Leffler, M. McKusick, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.3 BSD UNIX OS*. Addison-Wesley. 1989.

[LPL95] Moon Lee, Noah Prywes and Insup Lee. Automation of Analysis, Simulation and Understanding of Real-time Large Ada Software. *The First IEEE International Conference on Engineering of Complex Computer Systems*. IEEE Press. November 1995.

[Nav90a] Naval Sea Systems Command. PMS 412. *User Handbook for CMS-2 Compiler*. NAVSEA

0967-LP-598-8020. March 1990.

[Nav90b] Naval Sea Systems Command. PMS 412. *Program Performance Specification for CMS-2 Compiler*. NAVSEA 0967-LP-598-9020. March 1990.

[Sam95] A. Samuel, E. Sam, J. Haney, L. Welch, J. Lynch, T. Moffitt and W. Wright. Application of A Reengineering Methodology to Two AEGIS Weapon System Modules: A Case Study in Progress. *Proceedings of the Fifth Systems Reengineering Technology Workshop*. Montrey, CA. February 1995. pp. 69-79.

[Ste90] W. Richard Stevens, *UNIX Network Programming*. Prentice Hall Software Series. 1990.

[Sun90] Sun MicroSystems Inc. *Sun Network Programming Guide*. SunOS 4.1.1 Manual, Vol. 10. 1990.

[Univac] Sperry Univac. *AN/UYK-20 standard executive programmer study guide*. Sperry Univac, DSD Education, Saint Paul, Minnesota.

[Unisys] Unisys. *AN/UYK-44,Technical Description*. Unisys Corporation, Defense Systems, St. Paul Minnesota 555164-0525. November 1987.

이 문 근

1989년 미국, The Pennsylvania State University, Computer Science학과 졸업(이학사)

1992년 미국, The University of Pennsylvania, Computer and Information Science학과 졸업(이공학석사)

1995년 미국, The University of Pennsylvania, Computer and Information Science학과 졸업(이공학박사)

1992년 5월~1996년 1월 미국, Computer Command and Control Company(CCCC)에서 Computer Scientist로 근무

1996년 4월~현재 전북대학교 컴퓨터과학과 전임강사

관심분야: 소프트웨어 재·역공학, 실시간 시스템, 운영체제, 형식언어, 병렬함수언어, 컴파일러 등