실시간 소프트웨어의 조절적·단위적 이해 방법: ARSU(Architecture-based Software Understanding)와 SRE(Software Re/reverse-engineering Environment)

이 문 근†

요 약

본 논문은 매우 방대하고 복잡한 실시간 소프트웨어를 이해하기 위한 하나의 방법론과 도구의 개발 연구에 대하여 보고한다. 대부분 본 논문의 저자에 의하여 개발된 이 방법론과 도구는 ARSU(*Architecture-based Real-time Software Understanding*)과 *SRE(Software Re/reverse-engineering Environment)*이다. 재공학과정중에 실시간 소프트웨어의 이해한다는 것은 방대한 규모와 복잡성 때문에 일반적으로 매우 어려운 일이다. 그러나 이러한 어려움을 극복하기 위하여 본 논문에서는 architecture에 근거하여 구조적·기능적·행위적 측면에서 3차원적인 이해를 가능하게 한다. 이 방법을 통하여 실시간 소프트웨어를 점진적이며 체계적으로 재·역공학할 수 있게 한다. 첫째, 구조적인 관점은 부모-자식간의 관계에 기초한 상하 계층적으로 이루어진 소프트웨어의 전체적 구조, 그리고 명세와 알고리즘 뷰들을 통하여 관측할 수 있다. 여기에서 구조를 구성하는 기본단위는 SWU (*Software Unit*)이며 이 SWU는 특정 기준에 준하여 추출된다. 이 구조는 상하 또는 그 역방향으로 소프트웨어를 항해(navigation)할 수 있게 한다. 이는 소프트웨어에 대한 개요와 상세에 관한 정보를 분리하여 상호간에 연관성이 있게 보여준다. 구조의 어떤 단계, 즉 어떤 추상화 단계에서라도 소프트웨어에 대한 기능적·행위적 대한 정보를 얻을 수 있게 한다. 둘째, 기능적 뷰는 자료와 제어의 흐름, 입력과 출력, 정의와 사용, 변수와 참조 등을 보여준다. 이 뷰의 각 사항들은 소프트웨어에 대한 특정 기능 정보를 제공하여 준다. 셋째, 행위적 뷰는 상태도, IEL(interleaved event list) 등을 들 수 있다. 이 뷰는 소프트웨어에 대한 실행시 동적 성질을 보여준다. 이 뷰들 외에도 각 측면과 뷰들을 위한 기능, 접속, 주석, 코드 등의 다수의 서류들이 제공된다. 본 연구의 가장 큰 장점은 구조를 항해하면서 여러 차원의 정보를 추상화하거나 세부적으로 확장할 수 있는 기능이다. 이러한 기능들은 이러한 실시간 소프트웨어를 이해할 수 있는 토대를 마련해 준다. 그리고 이러한 장점은 재사용 가능한 요소를 체계적으로 식별하거나 검증할 수 있게 한다.

A Scalable and Modular Approach to Understanding of Real-time Software: An Architecture-based Software Understanding(ARSU) and the Software Re/reverse-engineering Environment(SRE)

Moon-Kun Lee†

† 정 회 원:전북대학교 컴퓨터과학과
논문접수:1997년 6월 4일, 심사완료:1997년 11월 12일

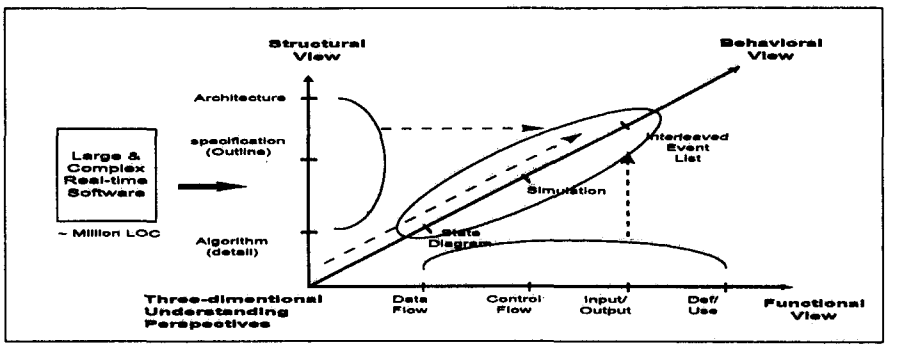
ABSTRACT

This paper reports a research to develop a methodology and a tool for understanding of very large and complex real-time software. The methodology and the tool mostly developed by the author are called the *Architecture-based Real-time Software Understanding* (ARSU) and the *Software Re/reverse-engineering Environment* (SRE) respectively. Due to size and complexity, it is commonly very hard to understand the software during reengineering process. However the research facilitates scalable re/reverse-engineering of such real-time software based on the architecture of the software in three-dimensional perspectives: *structural, functional,* and *behavioral* views. Firstly, the structural view reveals the overall *architecture, specification* (outline), and the *algorithm* (detail) views of the software, based on hierarchically organized parent-child relationship. The basic building block of the architecture is a *Software Unit* (SWU), generated by user-defined criteria. The architecture facilitates navigation of the software in top-down or bottom-up way. It captures the specification and algorithm views at different levels of abstraction. It also shows the functional and the behavioral information at these levels. Secondly, the functional view includes graphs of data/control flow, input/output, definition/use, variable/reference, etc. Each feature of the view contains different kind of functionality of the software. Thirdly, the behavioral view includes state diagrams, interleaved event lists, etc. This view shows the dynamic properties of the software at runtime. Beside these views, there are a number of other documents: capabilities, interfaces, comments, code, etc. One of the most powerful characteristics of this approach is the capability of abstracting and exploding these dimensional information in the architecture through navigation. These capabilities establish the foundation for scalable and modular understanding of the software. This approach allows engineers to extract reusable components from the software during reengineering process.

1. Introduction

Motivation

This paper reports a research to develop a methodology and a tool for understanding of very large and complex real-time software. The methodology and the tool mostly developed by the author are called *the Architecture-based Real-time Software Understanding* (ARSU) and the *Software Re/reverse-engineering Environment* (SRE) [1, 2, 3, 4, 5] respectively. Real-time software are usually very large in size (hundreds or thousands of program units, with hundreds of thousands or millions of lines of code) and very complex in concurrency, a/synchronization, communication, input/output (I/O) and time (hundreds or thousands processes communicating massively each other with real-time constraints). Due to size and complexity, it is commonly very hard to understand the software during reengineering process [2, 3].



(Fig. 1) An overview of a scalable approach for understanding of real-time system.

The Architecture-based Real-time Software Understanding(ARSU)

To solve this problem, the research in this paper provides engineers with a systematic approach to facilitate re/reverse-engineering such software, that is, ARSU.

As shown in (Fig 1), ARSU facilitates scalable and modular reverse-engineering of such real-time software based on the *architecturefe* of the software in three-dimensional perspectives: *structural, functional,* and behavioral views. These views are described as below:

i) **Structural view**: This view consists of three kinds of information: *architecture, specification, and algorithm*. First, the software is partitioned into two types of views: a specification view that contain the outline of the software, and a number of *algorithm* views that contains implementations of respective components in the specification view.

Next, each view is partitioned into software units (SWUs) based on program blocks and size of blocks. SWUs are clustered into groups based on degree of bindings (tuples) among them. The SWU is of the DoD standard for software specification [6] and a new standard for Software Design Documents [7]. Now, the architecture of the software can be constructed. The architecture is an overall skeleton of the software. It is hierarchically organized with based on *parent-child* relationship of SWUs. It facilitates navigation of the software in top-down or bottom-up way. At any level in the architecture, other related information beside the structural, such as the functional and the behavioral, can be abstracted or exploded in the direction of navigation. These information can be documented.

ii) **Functional view**: This view shows functionality of the software. This view includes data/control flow, input/output, definition/use, variable/reference information. In the view, the functionality is obtainable at the certain level of abstraction in the architecture. The functionality can be exploded further by traversing in depth. In addition, each functional information can be maintained in both specification and algorithm levels.

iii) **Behavioral view**: This view shows the dynamic properties of the software at runtime. It includes state diagrams, interleaved event lists, etc. State diagram is called Real-time State Machine for Reverse Specification (RSMRS). It is extended from Communicating Real-time State Machine (CRSM)[8] by the author. A RSMRS is obtained from an algorithm view. An interleaved event list is obtained by simulation of RSMRS. It shows a list of events and their relations among parallel threads at simulation time. It also contains related parent-child SWU relationships. Comparing with other approaches[9, 10, 11, 12], the view reveals the behavior of the software at certain architectural or abstraction level. It means that the behavior of the SWU is abstracted at certain architectural level. For detailed information, the behavior can be exploded further by traversing in depth.

Beside these views, there are a number of other documents: capabilities, interfaces, comments, code, etc. Capabilities here describe requirements for SWU. Interfaces describe external information to other SWUs. Comments include different kind of comments, such as the ones from program, compiler, linker, and debugger. These comments can be further extended while performing understanding.

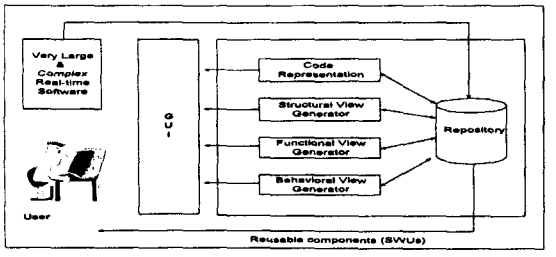
The advantage of the approach in the paper can be summarized as follows:

i) Scalability: Software understanding research usually focused on small programs in simple languages [13, 14, 15, 16, 17, 18]. The research in the paper has focused on software understanding of the very large real-life software with concurrent processes. The approach supports the scalability of large real-time software through incremental processing of modules. Here *scalability* means that each module or component of the software can be independently processed and integrated with previously input modules. Each module or file is repeatedly processed as described for under-

standing and integrated with previously processed modules to generate an overall architecture.

ii) **Dimensional architecture**: Software understanding research usually focused on understanding a slice of a large program which involved a limited view of the program [13, 14, 15, 16, 17, 18]. The approach in the paper has focused on understanding entire integrated real-time software. As stated, the approach provides engineers with an environment for understanding of the entire integrated real-time software at the different levels from the different perspectives, such as structural, functional, and behavioral. The approach allows a user to pursue understanding large and complex real-time software in a systematic top-down or bottom-up approach.

iii) **Reusability**: It is generally known that the vast majority of software labor costs is expended on maintenance of software [19]. Software re/reverse engineering is concerned with modernizing the legacy software for reuse, since there is a strong need for such reusable components to handle cost-effectively the size and complexity of the software in re/reverse engineering. The approach has focused on discovery of such functional components so that they can be used in different configurations. The partitioning of the software creates such functionally meaningful software units.



(Fig. 2) An overview of Software Re/reverse-engineering Environment (SRE).

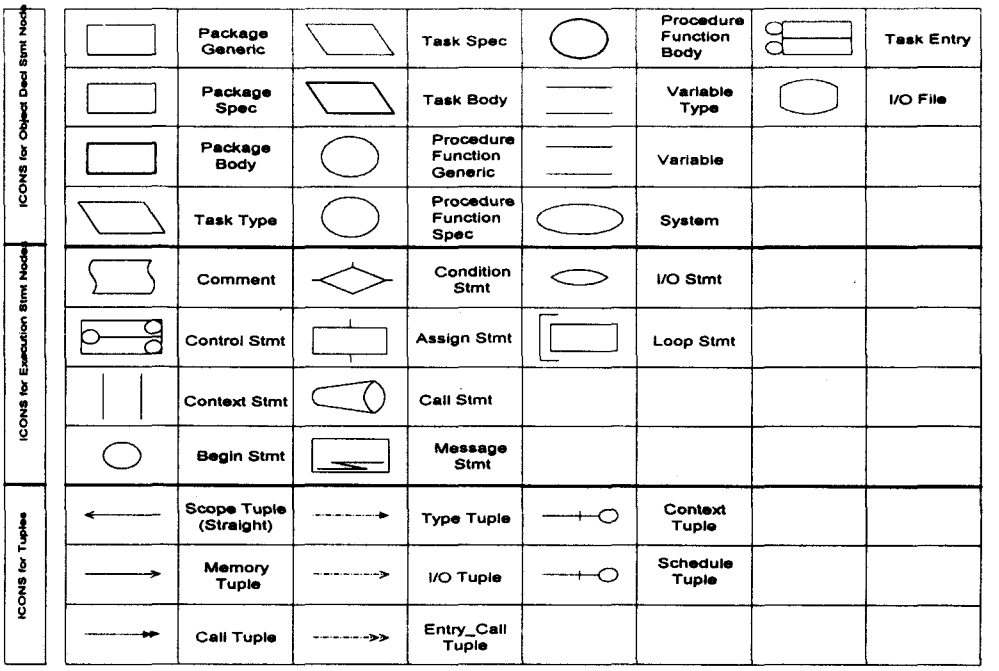
The Software Re/reverse-engineering Environment (SRE)

(Fig 2) illustrates the major components of the SRE that has been developed based on ARSU. The environment was built partly from the previously existing capabilities of the Software Reverse-engineering Environment (SRE) [1]. New components (structural, functional, and behavioral view generators) were developed to create the new environment by the author [2, 3, 5]. These components, except the code representation, are the generators that produce respective views as described. These components are self-explanatory.

Here the code representation is the process of representing the software in graphical format. In this process, the real-time software is represented by a graphical language, called the *Elementary Statement Language* (ESL) [2, 20] to facilitate diverse objectives of the approach in the paper. ESL is based on *Entity-Relation-Attribute* (ERA) graphs [21]. The repository contains information in a number of levels of abstraction. In the lowest level, each node represents an statement as an entity. The edges, called tuples, represent a variety of semantic relations between statement nodes. ESL not only represents full real-time features and the functional and behavioral information at different levels of abstraction, but also is used in translation from other programming languages.

In SRE, a *Graphical User Interface* (GUI) facilitates man-machine interactions during the analysis processes for software understanding. The user can interact with the environment through GUI. The user can view the following outputs of each view: ESL graphs, architecture, SWUs (definition, graphs, interfaces, capabilities, etc), data/control flow graphs, RSMRS diagrams, simulation (input/output), execution data, interleavings, etc. The inputs and outputs are stored in the environment repository. In the research, GUI is supported by DECDesign [22, 23].

The research reported in the paper is an innovative



(Fig. 3) Icons for ESL Statement Nodes and Tuples.

approach to the software understanding of very large and extremely complex real-time software: a scalable and modular understanding based on the architecture of the software with respect to structural, functional, and behavioral perspectives. This environment is one of the first comprehensive environments for analysis and understanding of such software.

Outline of the Paper

Outline of the paper is as follows. Section 2 describes the definition of ESL language and graphs. Sections 3, 4, and 5 describe the processes of the structural, functional, and the behavioral view generation respectively. Each of these processes is illustrated with a Producer-Consumer example. Finally, conclusion and future research are presented in Section 6.

2. Code Representation

This section overviews the process of ESL code generation from source real-time software. ESL is a graphical language to represent Ada software based on ERA model. In the ESL graph, the statements are represented as nodes, and the relations between statements as edges, called tuples.

ESL Node

There is an ESL statement for every statement in the software. It is represented as a node in the ESL graph. The ESL statements can be classified into two categories: 1) object declaration statements and 2) execution statements as shown in (Fig 3). In addition, the statements are divided also into block and terminal. The block and terminal statements reflect their intermediate and leaf positions in an ESL graph. The block statements can be represented by nodes in the ESL graph. The constituent statements in each block are children nodes of the block node. This leads to a hierarchical ESL graph. The relation, between nodes, forming the hierarchy is the *scope tuple*. Statement

nodes are linked hierarchically and among siblings by scope tuples to form the program tree graph. The scope tuple is described more in the next. The terminal statements form the leaves of the tree. A ESL statement node includes attributes of the respective statement: statement number, name, type, a trace to a source statement, etc. Each type of the grouped ESL statements is then visually represented by graphic icons as shown in (Fig 3). Thus, all the statements in the software are represented progressively in the graphic software model by nodes.

ESL Edge

Relationships between the ESL statement nodes are represented by edges, called *tuples*. The types of tuples are shown in (Fig 3). The figure shows, for each type of tuple, the statements that may be used as the source or target node of the tuple, as well as whether the source statement is an object or execution statement. Finally, it shows for each tuple the direction of the respective activity. The tuples are classified into two categories: 1) *scope tuple* and 2) non-hierarchical tuples.

The scope tuple is a relation that connects the constituents of each block statement node. An ESL graph contains edges which connect the nodes to form a tree. A tree branch from a node to another node at the same software hierarchical level means that the statement of the first node immediately precedes the statement of the latter node. A branch from a higher software hierarchical level block statement node to a chain of the next lower level nodes represents lexical containment. This type of edge is a scope tuple. Additional non-hierarchical relations (tuples) are then created between pairs of nodes to represent binary relations (tuples) between nodes. There are six additional types of tuples indicating respective relationships. These tuples are used in reordering and reorganizing code and for software understanding.

The non-hierarchical tuples represent: a relation between a statement that references or updates a variable and a statement that declares the variable, a relation between a procedure call statement and the procedure's declaration, a relation between a message call statement and the called entry point declaration, including synchronization calls, a relation between an I/O call and the respective device declaration, a relation between a generic or type declaration and respective instantiation statement, and a relation between a specification and its respective body declaration, or between with/use statement and the respective package declaration.

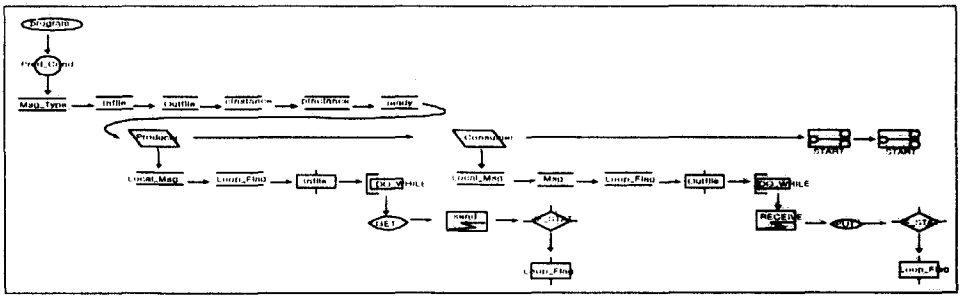
Each tuple has attributes, such as name, type, source and destination statement nodes, etc. Each tuple also has its respective icon as shown in (Fig 3).

Example

A Producer-Consumer example shown in (Fig 4) is used to illustrate understanding processes in the paper. The CHILL code [26] for the example is

```
Prod_Cons:
   NEWMODE MESSAGE_TYPE = CHARACTER;
   DCL cInstance INSTANCE;
   DCL pInstance INSTANCE;
   DCL Infile REF FILE;
   DCL Outfile REF FILE;
   DCL READY BUFFER CHAR(80);

   producer: PROCESS();
      DCL Local_Msg MESSAGE_TYPE;
      DCL Loop_Flag BOOL := TRUE;

      Infile := fopen("Infile_name", "r");
      DO WHILE Loop_Flag;
         Local_Msg := getc(Infile)
         SEND READY(Local_Msg);
         IF (Local_Msg = 'T') THEN
            Loop_Flag := FALSE;
         FI;
      OD;
   END producer;

   consumer: PROCESS();
      DCL Local_Msg MESSAGE_TYPE;
      DCL Msg MESSAGE_TYPE;
      DCL Loop_Flag BOOL := TRUE;

      Outfile := fopen("outfile_name", "w");
      DO WHILE Loop_Flag;
         Msg := RECEIVE READY ;
         Local_Msg := Msg;
         putc(Local_Msg, outfile);
         IF (Local_Msg = 'T') THEN
            Loop_Flag := FALSE;
         FI;
      OD;
   END consumer;

   START producer SET pInstance ;
   START consumer SET cInstance;

END Prod_cons;
```

(Fig. 4) CHILL Code for Producer-Consumer Example.



(Fig. 5) ESL Graph for Producer-Consumer Example.

designed and coded by the author. The example consists Procedure Prod_Cons. Procedure Prod_Cons declares variable type and local variables, and two processes (Producer and Consumer). Producer declares local variables, repeats reading the value of a variable from an external file, and sends the value to Consumer by communication until there is a termination signal ('T') from the external file. Consumer declares local variables, repeats accepting a message call from a caller, and stores the message in an external file until there is a termination message ('T') from Producer. The example is very small and is used only for illustration.

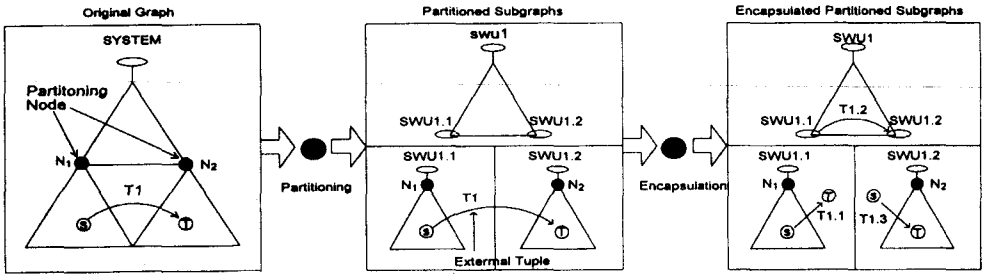
(Fig 5) shows the ESL graph with only scope edges of the Producer-Consumer example shown in (Fig 4). The nodes are denoted by icons for the respective statement types as shown in (Fig 3). The root of the graph is a SYSTEM node, named PROGRAM. Next child node is for tProcedure Prod_Cons. The child nodes of the Procedure Prod_Cons node are for variable type, variable declarations, Producer, Consumer, and start statements of the procedure. Producer and Consumer form sub trees. The child nodes of the task body nodes are for local variable declarations and a block of statements of the processes.

3. Structural View

This section describes the process of the structural view generation. This process is a way of abstracting an architecture of the real-time software, represented in the ESL. The discovery of architecture is accomplished by partitioning the software into smaller segments of code. The partitioning of the software is guided by criteria defined by a user. These partitioned segments of codes are called the *Software Units* (SWU). The SWU is of the DoD standard for software specification [6] and a new standard for Software Design Documents [7]. The architecture of the software is represented in a hierarchical tree of SWUs, where a descendant subunit shows an explosion of its parent unit. The architecture and SWUs are used for software understanding as well as for simulating execution of the software.

The discovery of the architecture has several purposes: i) to establish a foundation for software understanding, by providing the user with top-down progressively more detailed information on the software units, or with bottom-up more global information of the overall software, ii) to identify SWUs which may be selected to be tested, and iii) to obtain SWUs with a desired granularity that do not over-clutter the visualization. Here the granularity is determined by partitioning criteria. There are two type of the partitioning criteria: i) the types of the ESL block statements, and ii) the maximum target size range for the number of the ESL statement nodes in a SWU.

Partitioning Process



(Fig. 6) An Illustration of the Architecture Discovery

The first step to generate an architecture is the partitioning process. The process performs the partitioning of the ESL graph based on the criteria. The partitioning is performed as follows: i) Find the root node of the SWU. and ii) Generate an ESL subgraph of the SWU.

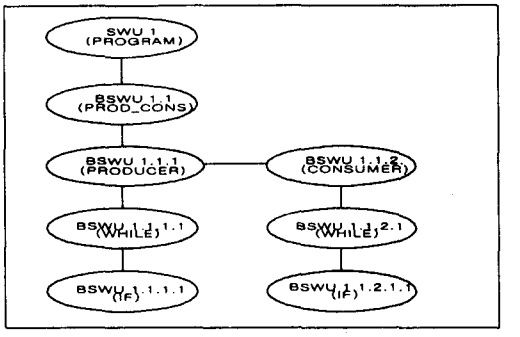
A subgraph is an explosion of a corresponding system node in the parent graph. The left side of (Fig 6) illustrates the partitioning process. The partitioned graphs are SWUs. These SWUs are hierarchically organized as a result of the partitioning. The structure of these hierarchically orgranized SWUs is the architecture of the software.

Encapsulation Process

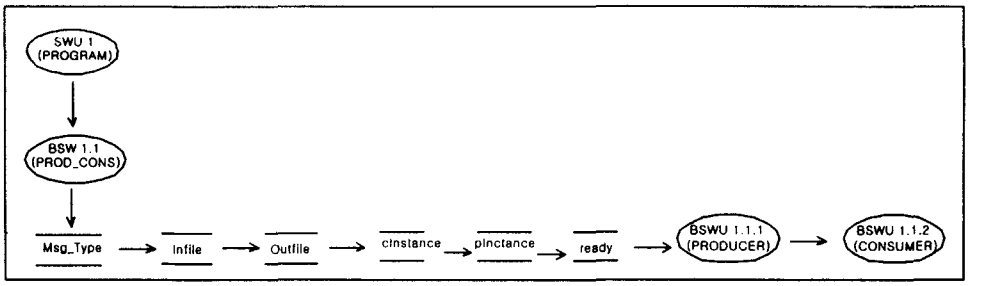
After the partitioning, the next step is to perform encapsulation of external tuples in each partitioned graph as shown in the right side of (Fig 6). The external tuple is a tuple whose source and destination nodes are not in the same subgraph. The encapsulation is a process of preserving the original information of the external tuple in the hierarchically partitioned graphs or SWUs. Firstlt, a new external node for the target node of the external tuple is created and a new tuple to this new external node as a target node is defined in the subgraph where the external tuple originates. Secondly, a tuple between the corresponding ancestor nodes of the external tuple is created in the common ancestor graph of the source and target nodes of the external tuple. Finally, a new external node for the target node or source node of the external tuple is created and a new tuple to this new external node as a target node is defined in other intermediate ancestor graphs between the source, target, and the common ancestor graphs.

The information of all descendent external tuples are preserved in the partitioned subgraphs through the encapsulation. For example as shown in (Fig 6), the information of an external tuple, Tuple T1, is preserved in Tuples T1.1, T1.2, and T1.3, where Tuple T1.2 is the abstraction of Tuple T1 at the SWU 1 level.

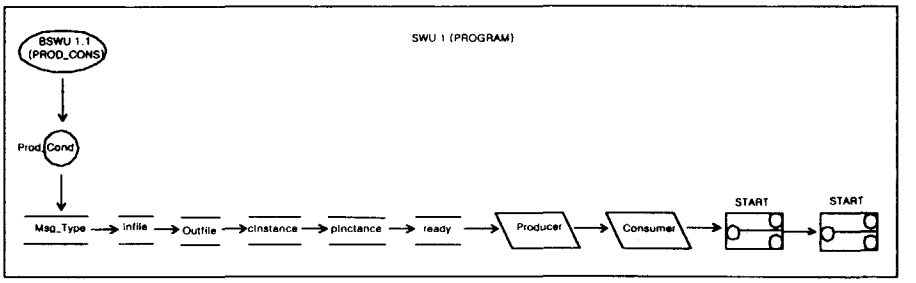
After completion of the partitioning and encapsulation processes, the following outputs are generated : i) an architecture of the software, and ii) each SWU with encapsulated tuples and external nodes. These architecture and SWUs are stored in the databases and can be retrieved by the user for visualization.



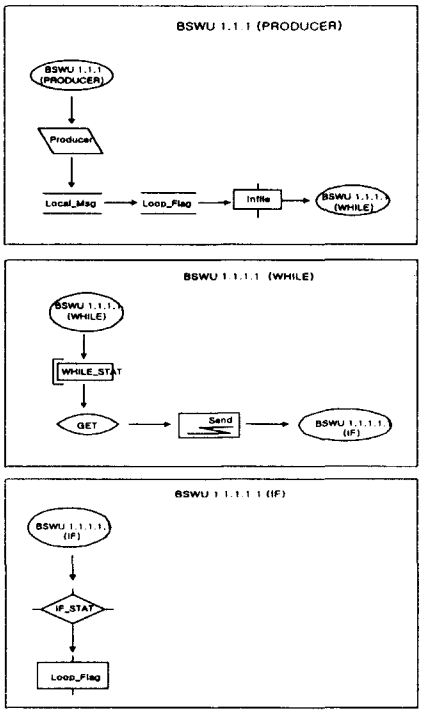
(Fig. 7) The architecture of Producer-Consumer Example



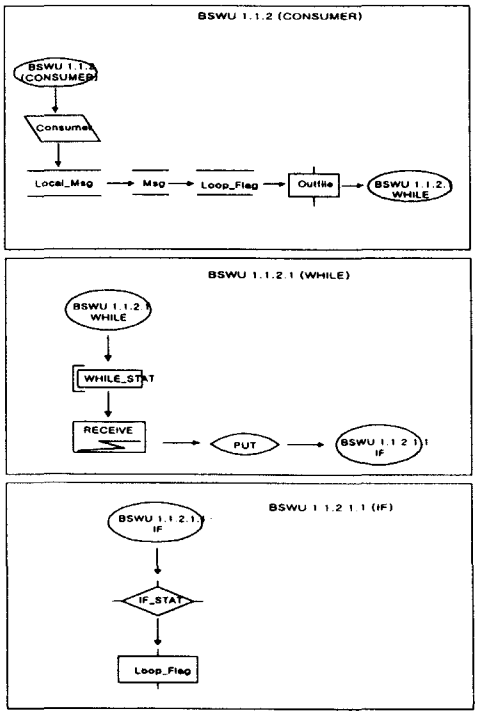
(Fig. 8) The specification view of SWU 1 of Producer-Consumer Example



(Fig. 9) The algorithm view for BSWU 1.1 of Procedure Prod-Cons



(Fig. 10) The algorithm Views of the BSWUs for the Producer Process



(Fig. 11) The algorithm Views of the BSWUs for the Consumer Process

(Fig 7), 8, 9, 10, and 11 are the outputs of the structural view generation for the Producer-Consumer example as shown in (Fig 5) and 6. (Fig 7) shows the architecture of the example. The SWUs are hierarchically organized in the architecture. (Fig 8) shows a single specification view of the ESL of the example. The ESL of algorithms (bodies) of the SWUs (BSWU) in the architecture are shown in (Figs 9), 10, and 11 respectively. Each BSWU is hierarchically organized based on containment relation.

4. Functional View

This section describes the process of generating functional views. The functional views show the functionality of the software with respect to SWUs along with the architecture. The functional views include data/control flow, input/output, definition/use, etc., as described below:

i) Data flow graph: Two types of data flow are obtainable. The first is the intra-SWU data flow. This flow is revealed by analyzing the definition of global/local variable and parameter and their references (read or write) in algorithm SWUs. This flow allows engineers to find inexecutable statements, unreferenced variables or parameters, or slicing information. The other is the inter-SWU data flow. This flow shows data dependencies among specification SWUs at certain level.

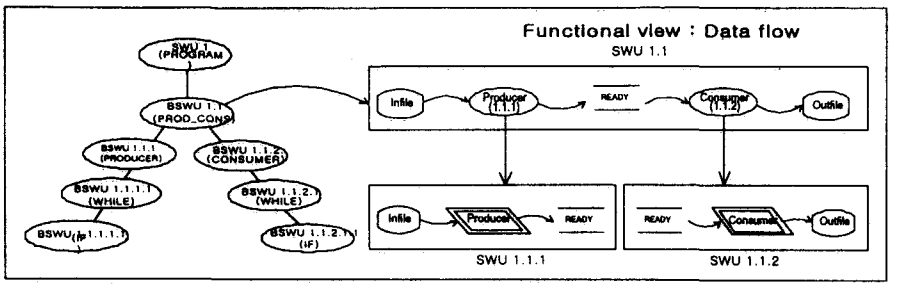
ii) Control flow graph: Two types of control flow are possible. The first is the intra-SWU control flow. This flow shows the order of execution of statements in algorithm SWU. The other is the inter-SWU control flow. It shows call dependencies among specification SWUs.

iii) Input/output: This is the interface information to SWUs.

iv) Definition/use: It shows the definitions of types/variables and their usage in the software.

In addition to the above features, one of the most powerful characteristics of the approach is that each feature of the functional view is hierarchically organized with respect to the architecture. The low level functionality is abstracted in the higher level and the higher level functionality is exploded in the lower level in detail. This capability allows engineers to navigate the architecture for understanding of functionality. In addition, such functionality is captured both in specification and algorithm SWUs.

To illustrate the functional view, (Fig 12) shows a number of data flow graphs of the Producer-Consumer example with respect to certain levels of abstraction along with the architecture. These graphs only contain data flow information in the specification. The left side of the figure is the architecture of the example. At the level of SWU 1.1, the data flow graph of the functional view is shown in the top of the right side of the figure. It is an abstraction of data flow specification for the respective algorithm SWU (BSWU). At the level of SWU 1.1.1 and 1.1.2, the data flow information is further exploded in detail



(Fig. 12) The abstraction of data flow graphs in the function view along with the architecture.

in two data flow graphs in the bottom of the right side of the figure. These graphs show which parts of components are directly inter-related with others.

5. Behavioral View

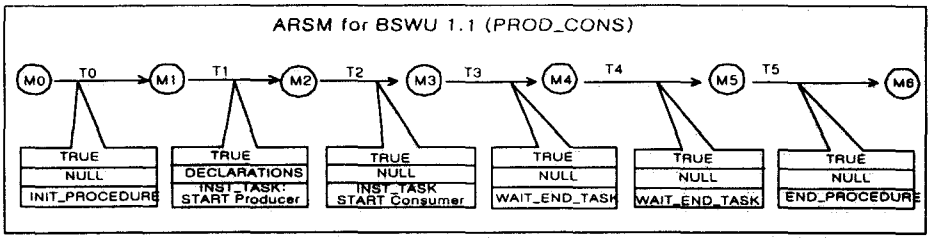
This section describes the process of the behavioral view generation. Behavioral view shows the dynamic properties of the software at runtime. It includes state diagrams, interleaved event lists, etc. This process is performed by the following steps. First, state machine is generated from an algorithm view. Next, an interleaved event list is obtained by simulating a state machine. It shows a list of events and their relations with other concurrently executable entities being simulated. In the following, the state machine and the simulation are described.

State Machine

This section describes the *Real-time State Machine for Reverse Specification* (RSMRS). The RSMRS is a new notion of a state machine for the real-time software, extended from the Communicating Real-time State Machine (CRSM) [8]. It represents algorithm views (BSWU). The objective of the RSMRS is to simulate the software. The generated data of the simulation execution are to be used in testing of the software.

The RSMRS consists of a set of *states* and a set of *transitions*. A state is a set of values of the variables referenced in the BSWU. The transition is a sequence of code fragment that may consist of three parts: a *CONDITION*, *STATEMENTs* or *DECLARATIONs*, and an *EVENT*. The CONDITION is the logical expression for the execution of a block of statements or the iteration of a loop in the software. The STATEMENTs may be a sequence of assignment statements. The DECLARATIONs are variable declarations. The EVENT is a statement that requires input, produces output, interacts with other tasks, calls a procedure/function, etc. The execution of a statement in the BSWU changes the value of a variable. The change in the values of variables implies a change of the state in the RSMRS. The basic approach of the state transition in the RSMRS is that a state precedes the occurrences of a CONDITION or EVENT statement. This approach reduces the number of states and transitions. The transition relation from one state to another means that a CONDITION (if any) is satisfied, a set of STATEMENTs or DECLARATIONs are executed, and an EVENT may have occurred. In the graph for the RSMRS, a state is represented as a node and a transition as an edge. Each state and transition have a unique ID and name.

A RSMRS represents a concurrently executable entity. Communication between them is performed in one-to-one synchronous manner. Compared with other real-time state machines [9, 10, 11, 12], the innovative features of RSMRS are: i) RSMRS is organized hierarchically to reduce the size of a component state machine to an understandable level, ii) it represents the full real-time features, and iii) it represents the transfer of execution control. These

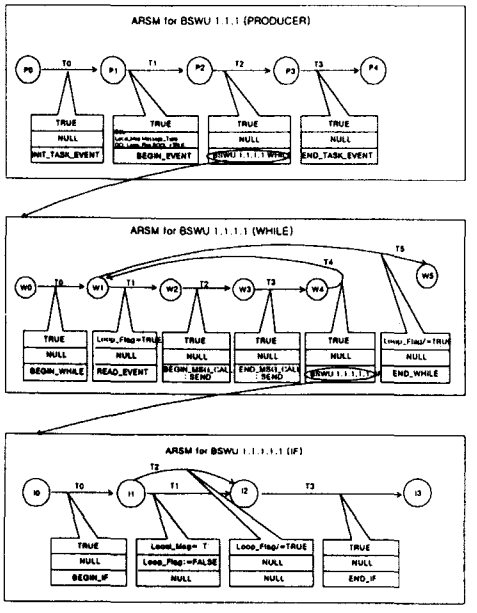


(Fig. 13) RSMRS for BSWU 1.1 of Prod_Cons Procedure Body

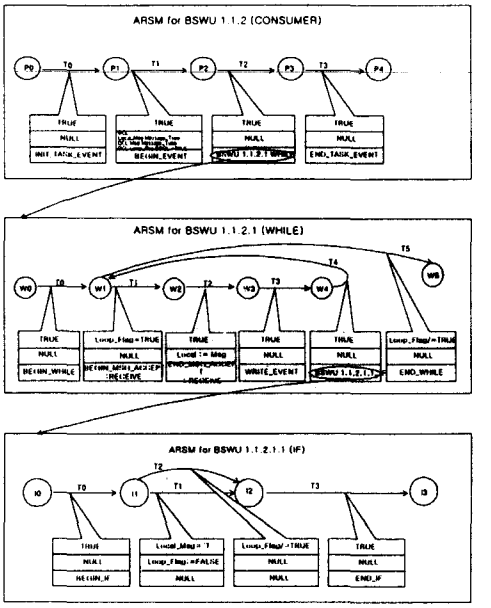
features allow to represent very large and complex real-time software in a state machine. In SRE, a RSMRS is visualized through the graphical user interface.

The PROD_CONS, PRODUCER, and CONSUMER RSMRSs are shown in (Fig 13), 14, and 15 respectively. The BSWUs for these state machines are shown in (Fig 9), 10, and 11 respectively. In the following, each RSMRS is described.

i) PROD-CONS PROD-CONS RSMRS : The PROD-CONS RSMRS is shown in (Fig 13). State M_0 is an initial state and State M_6 is the final state. Note that the EVENTs of Transitions T_0 and T_5 are the INIT_PROC_EVENT and END_PROC_EVENT respectively. These EVENTs are for the initializing and terminating the execution of the BSWU 1.1 (PROD-CONS) represented in the PROD-CONS RSMRS. The STMT/DCL of Transition T_1 is for the declarations of type, variables and processes. The EVENTs of Transitions T_1 and T_2 are for instantiations of Producer and Consumer processes. The EVENTs of Transitions T_4 and T_5 are to wait for the termination



(Fig. 14) RSMRSs for BSWUs of Producer Process



(Fig. 15) RSMRSs for BSWUs of Consumer Process

of the child processes : Producer and Consumer.

ii) PRODUCER RSMRS : The PRODUCER RSMRS is shown in (Fig 14). It has two child RSMRSs : the WHILE and the IF RSMRSs. The EVENTs of Transitions T_0 and T_3 are to initialize and terminate the execution of Producer represented in the PRODUCER RSMRS. The STMT/DCL of Transition T_1 is for declaration of the variables : Local_Msg and *Loop_Flag*. Note that the EVENT of Transition T_2 is for the BSWU 1.1.1.1 (WHILE). The EVENT implies jumping into State W_0 of the WHILE RSMRS. The transition starts from this state. At the time of reaching the final state of the WHILE RSMRS, the transition occurs to State P_4 of the PRODUCER RSMRS.

The WHILE RSMRS is shown in the same figure. The EVENTs of Transitions T_0 and T_6 are for the beginning and ending of the *while* block statement. The CONDITIONs of Transitions T_1 and T_5 are for the continuation and termination of the loop. The EVENT of Transition T_1 is to read data from a file.

The EVENTs of Transitions T_2 and T_3 are for the entry call to Task Consumer represented in the CONSUMER RSMRS. Note that the EVENT of Transition T_4 is for the BSWU 1.1.1.1.1 (IF). The EVENT implies jumping into State I_0 of the IF RSMRS. The transition starts from this state. At the time of reaching the final state of the IF RSMRS, the transition occurs to the next state of the WHILE RSMRS, State W_1.

The IF RSMRS is shown in the same figure. The EVENTs of Transitions T_0 and T_3 are for the beginning and ending of the *if* block statement. The CONDITIONs of Transitions T_1 and T_2 are for the then and else block statements of the *if* statement. The STMT/DCL of Transition T_1 assigns a new value to a variable.

iii) CONSUMER RSMRS: The CONSUMER RSMRS is shown in (Fig 15). It is similar to the Producer RSMRS except the *RECEIVE* event instead of SEND event. The figure is self-explanatory.

RSMRS Simulator

This Section describes the Simulator for RSMRS. RSMRS represents a concurrent real-time software unit. The objective of the simulator is to simulate execution of real-time software in the representation of RSMRS.

The simulator requires the following inputs: i) the ID of the root SWU, ii) input communications and external variable values (test input), and iii) a list of the relative *CPU speeds* for each respective task in the SWU. A CPU is assumed dedicated to each task. The simulation of the tasks with different CPU speeds can validate the correctness of the output independently on CPU speeds.

The simulator constructs a *Task Control Block Table* (TCBT). The objective of the TCBT is to control execution of the tasks during the simulation. The entities simulated in the simulator are tasks (processes), which are concurrently executable. The TCBT is similar to the *Process Control Block* (PCB) in the Unix Operating System [24, 25]. The TCBT contains the simulation information about tasks. The simulation information in the TCBT includes ID, name, task state, simulation state, and other simulation-related data.

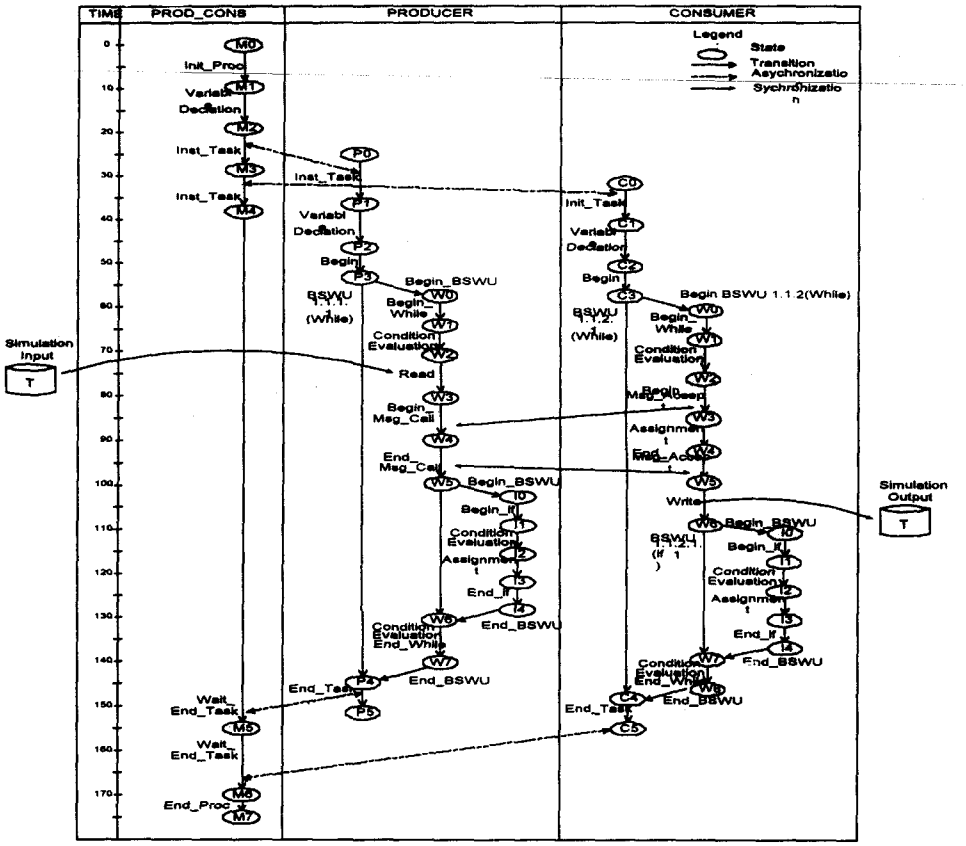
The approach in the simulator is as follows. The simulator selects a task with the earliest next simulation state transition time from the TCBT. It performs the simulation of the selected transition. It updates the TCBT block of the selected task. It iterates the selection process until all tasks in the TCBT are completed or the simulation time expires.

Upon completion of the simulation, it provides the user with a list of the outputs. These outputs include: i) an *interleaved event list*, which shows the traversed interleaved execution path and conditions, ii) output communications and external variable values (test output), and iii) the simulation input to output delays in standard simulation time units.

An interleaved event list is shown in (Fig 16). It contains all events for each task that occurred during the simulation. The events in the list are presented in the chronological order. The event in the list consists of the time of the occurrence, condition, and the type of the event, and other corresponding a/synchronous event, if any. The list in (Fig 16) illustrates the traversed interleaved execution path and conditions. Such lists for selected tasks can be used to verify the properties and the correct execution behaviors of the tasks.

Besides these, other simulation related outputs are obtainable. These are the waiting time, the response time, the execution time, or even the utilization of the virtual processor on which each task has been simulated. The information is usable to determine the resource requirements of the SWU. The information also can be used for effective allocation of the tasks to the hardware.

The outputs or simulation execution data can be used by the user to understand the simulated execution behavior of the tasks. The behavior



(Fig. 16) A Interleaved event list.

includes the state machine traversed path of the simulated execution of the tasks, interactions among the tasks through a/synchronous events, etc. The outputs can be further used to verify some properties. The properties of interest include the number of tasks active at the specific simulation time, the number of communications occurred between the tasks, etc. The output also can be used to verify the correct execution of the tasks. The behavior includes the timing requirements for communications between the tasks or the simulation input-to-output delays, etc.

6. Conclusion and future research

This paper reported a research for understanding of very large and complex real-time software. The approach in the research facilitates scalable re/reverse-engineering of such real-time software based on the architecture of the software *in* three-dimensional perspectives. The perspectives include *structural*, *functional*, and behavioral views as illustrated in (Fig 1).

One of the most powerful feature in the research is the capability of extracting and exploding such information in the architecture. These capabilities establish the foundation for understanding of the very large and complex real-time software during reengineering process. Through the different levels of software understanding in the different perspectives, the approach in the research extracts reusable components

from the software during reengineering process.

As a future research, the simulation part of SRE has to be implemented. This will be accomplished in near future.

The research reported in the paper is an innovative approach to the software understanding of very large and extremely complex real-time software: a scalable and modular understanding based on the architecture of the software with respect to structural, functional, and behavioral perspectives.

Acknowledgement

This research is supported in part by research grant from the Center for Advanced Image and Information Technology (CAIIT) of the Chonbuk National University, the Korea Science and Engineering Foundation (KSEF), and the System Engineering Research Institute (SERI).

References

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

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

[3] 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.

[4] Noah Prywes. *Reverse Engineering of CMS-2 to Ada.* Reengineering Workshop. NSWC, MA. August 1992.

[5] Noah Prywes, Giorgio Ingargiola, Insup Lee, and Moon Lee. Reengineering Concurrent Software into Ada. *The Proceedings of the 4th Systems Reengineering Technology Workshop.* Montrey, CA. February 1994.

[6] DoD. *Defense System Software Development.* DOD-STD-2167A. September 1988.

[7] DoD. *Military Standard Software Development and Documentation (Draft).* DOD Harmonization Working Group. December 1992.

[8] A. Shaw. Communicating Real-Time State Machines. *IEEE Transactions on Software Engineering.* Vol. 18, No. 9. September 1992. pp. 805-816.

9. D. Harel. Statecharts: A Visual Formalism for Complex System. *Science of Computer Programming.* Vol. 8. 1987. pp. 231-274.

[10] S. Jahanian and A. Mok. *Modechart: A Specification Language for Real Time Systems.* IBM Technical Report: RC 15140, November 1989.

[11] A. Shaw. Communicating Real-Time State Machines. *IEEE Transactions on Software Engineering.* Vol. 18, No. 9. September 1992. pp. 805-816.

[12] I. Kang and I. Lee. State Minimization for Concurrent System Analysis Based on State Space Exploration. *Proceedings of Conference on Computer Assurance.* June 1994. Gaithersburg, MD.

[13] T. Corbi. *Program Understanding: Challenge for the 1990s.* IBM Systems Journal. Vol. 28, No. 2. 1989. pp. 294-306.

[14] M. T. Harandi and J. Q. Ning. Knowledge-Based Program Analysis. *IEEE Software.* Vol. 7, No. 1. January 1990. pp. 74-81.

[15] J. Hartman. Understanding Natural Programs Using Proper Decomposition. *The Proceedings of the 13th International Conference on Software Engineering.* May 1991. pp. 62-73.

[16] J. A. Ricketts, J. C. DelMonaco, and M. W. Weeks. Data Reengineering for Application Systems. *The Proceedings of Conference on Software, Maintenance.* IEEE Computer Society Press, Los Alamits, CA. 1989. pp. 174-179.

[17] D. J. Robon, K. H. Bennett, B. J. Cornelius, and M. Munro. Approaches to Program Comprehension. *The Journal of Systems and Software*. Vol. 14. February 1991. pp. 79-84.

[18] Franz J. Polster. Use of Software Through Generation of Partial Systems. *IEEE Transactions on Software Engineering*. Vol. SE-12, No. 3. March 1986. pp. 402-416.

[19] J. Ahrens, N. Prywes, and E. Lock. Software Process Reengineering: Toward a new Generation of CASE Technology. *Journal of Systems and Software*. July 1995.

[20] 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.

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

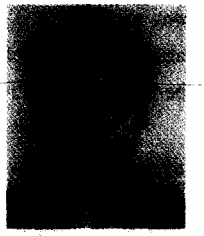
[22] Digital Equipment Corporation. *DECDesign: User's Guide*. DEC, Maynard, MA, Part No. AA-PABRE-TE. October 1991.

[23] Digital Equipment Corporation. *Digital: CDD/Repository: Using CDD/Repository on VMS Systems*. DEC, Maynard, MA. Part No. AA-P51KA-TE. October 1991.

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

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

[26] Electronics and Telecommunications Research Institute (ETRI), CHILL Programming Language, ETRI, Korea, 1993.



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