# Reverse Engineering of Computer-Based Navy Systems

**Lonnie R. Welch**
**Guohui Yu**
**Binoy Ravindran**
**Franz Kurfess**
**Jorge Henriques**
Department of Computer and Information Science
New Jersey Institute of Technology
Newark, NJ 07102
e-mail: welch@vienna.njit.edu
Phone: (201)596-5683
Fax: (201)596-5777 [*]

**Mark Wilson**
The Naval Surface Warfare Center
Dahlgren Division, White Oak Detachment
Code B44
Silver Spring, Maryland 20903-5640
**Antonio L. Samuel**
**Michael W. Masters**
The Naval Surface Warfare Center
Dahlgren Division
Dahlgren, VA 22448

[*] Supported in part by The U.S. NSWC (N60921-93-M-1912 and N60921-94-M-G096), by the U.S. ONR (N00014-92-J-1367), and by the State of New Jersey (SBR-421290).

# Abstract

The financial pressure to meet the need for change in computer-based systems through evolution rather than through revolution has spawned the discipline of reengineering. One driving factor of reengineering is that it is increasingly becoming the case that enhanced requirements placed on computer-based systems are overstressing the processing resources of the systems. Thus, the distribution of processing load over highly parallel and distributed hardware architectures has become part of the reengineering process for computer-based Navy systems.

This paper presents an intermediate representation (IR) for capturing features of computer-based systems to enable reengineering for concurrency. A novel feature of the IR is that it incorporates the mission critical software architecture, a view that enables information to be captured at five levels of granularity: the element/program level, the task level, the module/class/package level, the method/procedure level, and the statement/instruction level. An approach to reverse engineering is presented, in which the IR is captured, and is analyzed to identify potential concurrency. Thus, the paper defines *concurrency metrics* to guide the reengineering tasks of identifying, enhancing, and assessing concurrency, and for performing partitioning and assignment. Concurrency metrics are defined at several tiers of the mission critical software architecture. In addition to contributing an approach to reverse engineering for computer-based systems, the paper also discusses a reverse engineering analysis toolset that constructs and displays the IR and the concurrency metrics for Ada programs. Additionally, the paper contains a discussion of the context of our reengineering efforts within the United States Navy, by describing two reengineering projects focused on sussystems of the AEGIS Weapon System.

# 1 Introduction

A computer-based system has many characteristics, including performance, timeliness, availability, dependability, safety and security. Furthermore, such a system typically performs many related functions concurrently, interacts with the environment and many human operators and/or clients simultaneously, consists of many interconnected processing elements, contains many millions or tens of millions of lines of code, takes years to develop from first concept formulation to final deployment, and has development costs of many tens or hundreds of millions of dollars. Computer-Based systems generally address nontransient requirements that simply cannot be addressed with simpler solutions. Thus they tend to be characterized by long life cycles, often spanning decades. During such extended life cycles, change is inevitable in many dimensions, such as operational environment, system requirements, and technology base. Because of the time and cost of development of computer-based systems, and because of the infrastructure which includes highly trained personnel, hardware and support tools, documentation, test procedures, and many other components needed for their development and continued support once deployed, there is enormous financial pressure to meet the need for change through evolution rather than revolution. This need has spawned the discipline of reengineering, the systematic application of methodology and tools for managing the evolutionary transformation of existing computer-based systems to encompass new or altered requirements and to transport such systems into new environments and onto new technology bases.

It is increasingly becoming the case that the increased requirements placed on computer-based systems are overstressing the processing resources of the systems. Thus, designs and implementations are being reengineered to exploit highly parallel and distributed hardware. While computer-based systems of the previous generation employed some parallel processing, they seldom used more than a few processors. In contrast, modern computer-based Navy systems that use hundreds of processors are being prototyped. Since computer-based systems were developed for small scale parallelism in programming paradigms that supported little or no expression of parallelism, the exploitation of the tremendously increased parallelism is a challenge that must be addressed during reengineering.

In conjunction with concurrency enhancement for the accommodation of enhanced requirements, the Navy's reengineering efforts are employing modern software engineering principles to reduce the costs of design, implementation, testing, verification, and maintenance. Layering of software components is one technique being used to address these concerns. When a system is constructed by layering, the benefits include encapsulation and information hiding (i.e., loose coupling of software components) [20], abstraction (highly cohesive modules), and ease of understandability. Another benefit of layering is the simplification of analyses for: (1) concurrency [37, 39, 33, 36, 42, 32, 30], (2) timing properties [33, 31], and (3) dependability [11].

This paper describes a reengineering process that is appropriate for mission critical systems, such as the U.S. Navy's AEGIS system [23, 24]. Characteristics of past, present, and future computer-based Navy systems are described in Section 2. Also described in Section 2 are two reengineering projects [18] in which the authors participated. Section 3 describes a reengineering process which has been developed for transitioning Navy systems to meet the challenges of exploiting the technology of the present and of the future. Section 4 discusses reverse engineering for mission critical systems: the reverse engineering process is presented and the intermediate representation (IR) that is used to capture important software features is discussed. The reverse engineering efforts of the NJIT/NSWC team have resulted realizations of software tools to

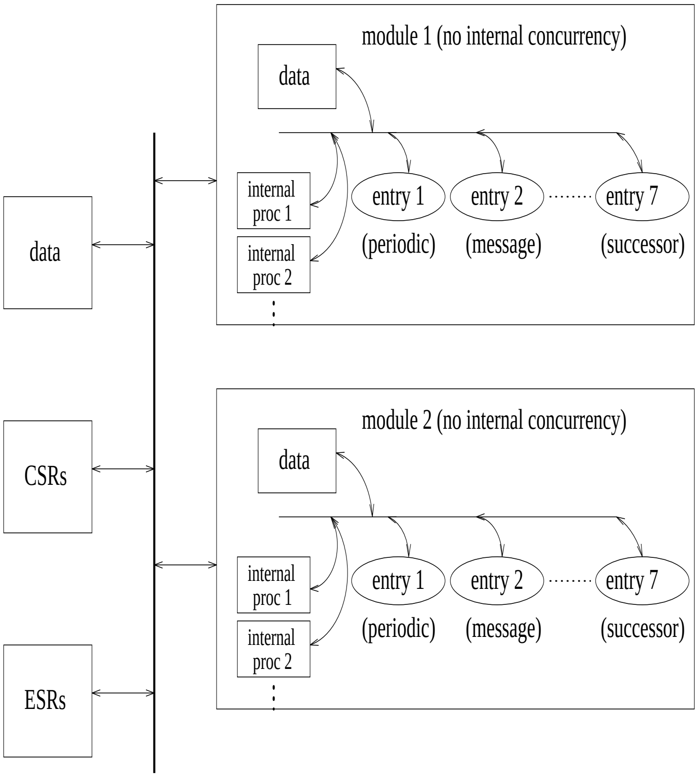


Figure 1: A typical model of previous generation software.

extract the IR from Ada programs. In Section 5, the techniques and algorithms embodied in the tools that construct the intermediate representation are presented. Additionally, graphical and hypertextual techniques for navigating the IR are described in Section 6. Finally, the use of the IR for the definition and computation of concurrency metrics for components within such an architecture is discussed in Section 7.

# 2 Computer-based Navy Systems: Past and Present

In this section the properties of computer-based Navy systems are described. The properties of systems that are being reengineered presently are reviewed, as well as the desired products of the reengineering process. The section concludes with an overview of two reengineering projects performed on the U.S. Navy's AEGIS Weapon System.

## 2.1 Properties of Legacy Systems

Computer-Based Navy systems built more than one decade ago exhibit certain trends, reflecting the previous state-of-the-art in the areas of software, hardware, and operating system technology. A typical software paradigm observed is one in which procedures are combined to form a module (see Figure 1). A module may contain exported operations (callable from within or without the module) and internal operations (callable only from within the module). In addition to containing operations, each module may contain data accessible only by its operations. Each exported operation of a module is termed a module entry, and serves as a means of manipulating the module's internal state. An important consideration during reengineering is that assembly language appears frequently in the code bodies of entries, since module development languages (such as CMS-2) provide no easy way to control hardware device accesses. In addition to the user-defined modules, a system may contain global data (tables) that are accessible by the operations of any module. There is also a set of common service routines (CSRs) and executive service requests (ESRs), callable from any operation.

It is beyond the scope of this paper to describe any details of the AN/UYK-xx (AN/UYK-20, AN/UYK-44, AN/UYK-7, AN/UYK-43) or AN/AYK-14 computers. Likewise it is not necessary to dwell on the differences between various versions of the Navy standard CMS-2 programming language (CMS-2M, CMS-2L, etc.) or the details of assembly languages such as ULTRA-16 or Macro, etc. Suffice it to say that present Navy combat systems rely on these military standard computers and languages [26, 27, 28]. Ships under construction presently are being built with UYK hardware and will run computer programs written in these languages for years to come. Likewise, Navy aircraft generally have AYK-14 computers onboard. The programs which run on these systems are invariably a combination of CMS-2 and assembly language.

Assembly code is often embedded within a CMS-2 program. This so-called direct or in-line code may represent a small fraction of code in a given program or an entire system's computer program may be written in assembly with a few lines of CMS-2 wrapped around it at the beginning and at the end. The percentage of assembly code generally varies with the performance requirements of the system and memory available. Hence, shipboard systems are likely to have a smaller percentage of assembly code than is found in aircraft systems.

Reengineering Navy tactical systems requires an approach that addresses both software and hardware. Much of the time there is also humanware that needs to be carefully considered during system design. However, that will not be addressed here except to point out that it is much more than a human-machine interface, display, or ergonomic issue. This is the case since it is generally undesirable to deploy powerful weapon systems that detect and fire without human intervention (even though it is technically feasible to fully automate many systems). The speed and accuracy of this human intervention should be a an integral part of the overall system design.

Concurrency and timing properties are stated by defining periodic module entries. Each of these executes once per period and may have a deadline (by which any particular execution of the entry must complete). At most one entry may be active within a module at any time (i.e., modules are monitors). A computer-based system is composed of many independent activities (or threads of control), which are implemented via calls to module entries, to ESRs and to CSRs, and which may directly access global tables. Due to the lack of layering, all modules, tables, CSRs and ESRs are visible to each activity.

There are several reasons for migrating from the aforementioned paradigm. As the functionality of computer-based systems increases, it is desirable to increase the concurrency in order to meet the timing requirements. Thus, systems should be portable to different hardware platforms. The frequent use of assembly language to implement entries significantly decreases portability. There are many other problems with the assembly language programs, many of them a result of the ability of clever programmers to promulgate a variety of potentially "dangerous" programming techniques in assembly code. Difficulties that may be encountered in assembly code include: widespread use of global data, hard coded data references, reentrant code, self modifying code, variable aliasing, pointers and other forms of indirection, overlays, and built in delays predicated on known performance of a specific processor. Furthermore, the lack of concurrency in system designs makes it difficult to exploit a large parallel processor, and the use of programming constructs like pointers makes the automatic analysis of concurrency troublesome. Also, the flat module hierarchy structure permits the access of global data structures by every procedure, leading to inefficiencies due to synchronization of accesses to such structures. Another deficiency is the lack of usage of modern software engineering concepts like abstract data types and objects, and generic (as can be implemented by Ada packages and C++ classes). The use of such constructs increases layering, improves reusability, and simplifies development, reengineering, timing analysis, and concurrency extraction [20, 37, 25]. The hardware model makes it impossible to achieve the large scale concurrency necessitated by the massive capabilities of modern software systems. Additionally, the degree of concurrency managed by the systems is very low, and modern concurrent execution paradigms such as asynchronous remote procedure call [37] are not supported. Another void in the systems is in the run-time support for efficient use of modern software engineering constructs such as abstract data types and abstract data objects.

## 2.2 The Target Paradigms of Reengineering

Many of the shortcomings of "yesterday's" system development paradigms are being overcome by employing modern paradigms. Layering, loose coupling, reuse [20, 22], high cohesion, encapsulation, and information hiding are facilitated by the proper use of programming constructs such as abstract data types (ADTs) and abstract data objects (ADOs). The ability to define generic ADTs and ADOs enables the development of parameterized abstractions, resulting in increased reuse and in a high payoff when such a component is produced by reengineering (since the cost of reengineering a component can be amortized over multiple uses of the component). The specification of concurrency can be performed in modern languages such as Ada by defining tasks within ADT or ADO modules. The ability to lexically nest modules reduces the visibility of data structures to only those needing to access them, thus lowering module coupling, simplifying analysis of concurrency, and leading to systems with fewer bugs. The amount of direct (assembly) code can be minimized in modern systems, leading to increased portability.

As an example, consider an AEGIS cruiser with several doctrine regions. An AEGIS cruiser is responsible for defending a fleet of ships. One of its many functions is to detect all objects (tracks) of interest, to classify detected tracks as friend or foe, to further classify into a particular kind of entity, and to respond appropriately if a entity is classified as a threat. Entering a doctrine region triggers detection, classification and possible response. As shown in Figure 2, doctrine regions are geometric areas in which certain actions may be taken. The engageability region is one in which tracks can be engaged with countermeasures. The early detection region surrounds the AEGIS cruiser, and is the region in which tracks are detected and classified. The Figure depicts a ship being protected by the AEGIS cruiser; thus, there is also an early detection region surrounding the other ship. The tight zone is a region in which engagements are prohibited (e.g., a commercial airway).

The AEGIS cruiser with doctrine region processing can be implemented with ADTs as represented in Figure 3, which shows the ADT instances used in the implementation, as well as call relationships among the instances. The track file is implemented as a list of tracks. A track is an ADT implemented as a queue of the last $n$ snapshots of the track's state.

An abundance of concurrency is available when the asynchronous remote procedure call

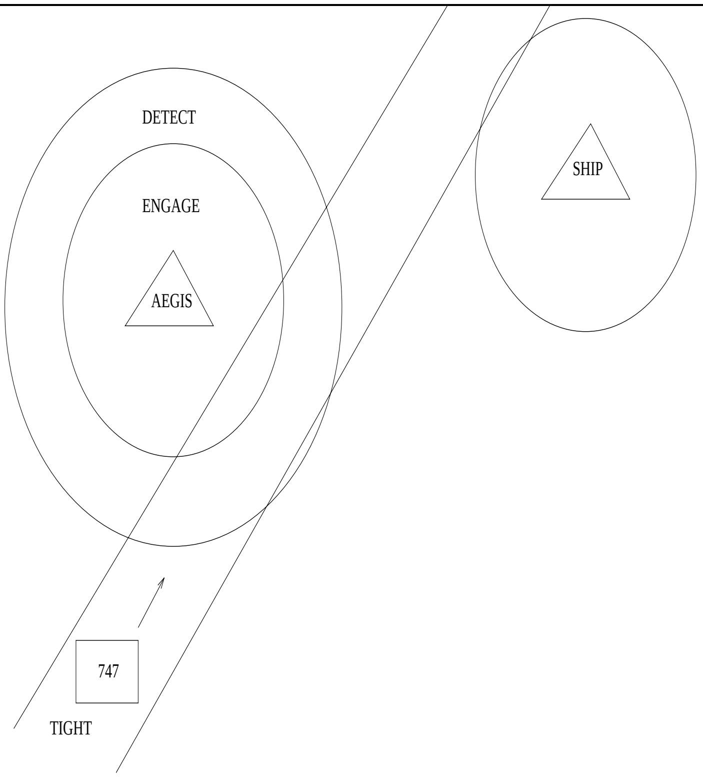


Figure 2: Sample doctrine regions.

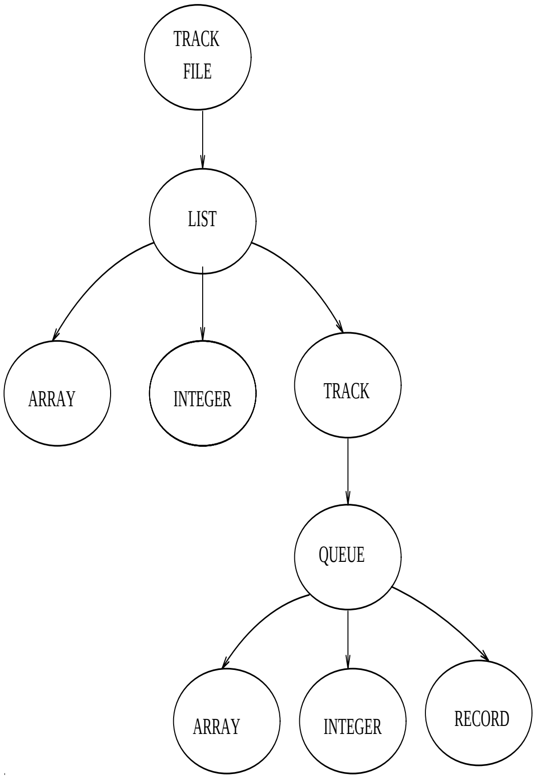


Figure 3: A design of doctrine processing software (ADT paradigm).

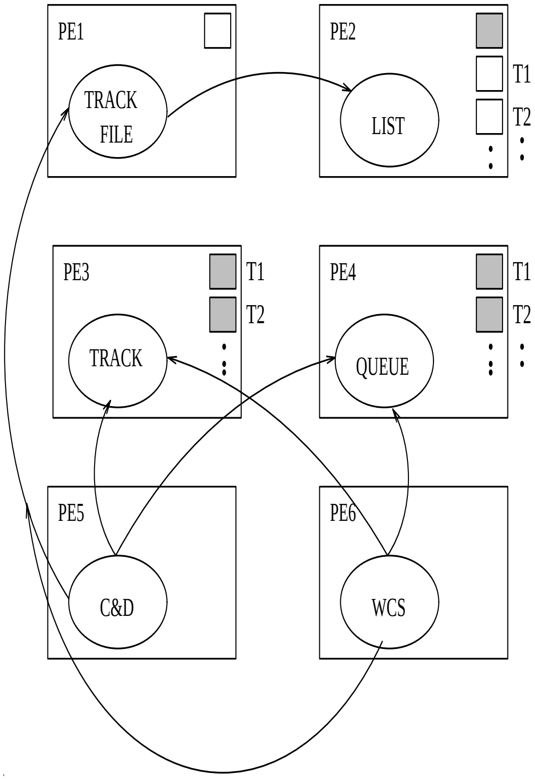


Figure 4: A possible assignment of ADT modules to processors.

model of parallel execution is applied to programs constructed from ADT and ADO modules (in addition to task-based concurrency). Instances (consisting of code and possibly state) are statically assigned to PEs (multiple instances may reside on the same PE). Instances' operations are invoked by sending call messages between PEs. To hide the latency of a remote call, an operation is permitted to continue execution until it attempts to access a "locked" variable (this model of concurrent execution is termed asynchronous remote procedure call, or ARPC [37]). A variable is automatically locked when it is passed as a parameter to a call and is unlocked upon return of the call. An operation attempting to access a locked variable must wait for a remote call to return before retrying the access. ARPC can achieve concurrency at multiple levels in the abstraction hierarchy. Thus, potential concurrency within a program increases with the number of levels of abstraction, and the model encourages development of highly cohesive, loosely coupled modules.

With the increase in potential concurrency comes the added complexity of exploiting the concurrency. Software components must be partitioned/clustered according to some binding relationships (such as communication, concurrency or shared data access), and the clusters assigned to processors in a way that causes efficient utilization of hardware resources and simultaneously obeys system constraints [32, 33, 34, 35, 29]. For example, the ADTs implementing the software for the AEGIS cruiser with doctrine regions could be assigned to processors as shown in Figure 4.

The assignment shown in Figure 4 will yield concurrency when ARPC is used as the execution model. However, the C&D and the WCS elements will serialize when they need to access tracks, even if they need different tracks. This is because all tracks are managed by the same code

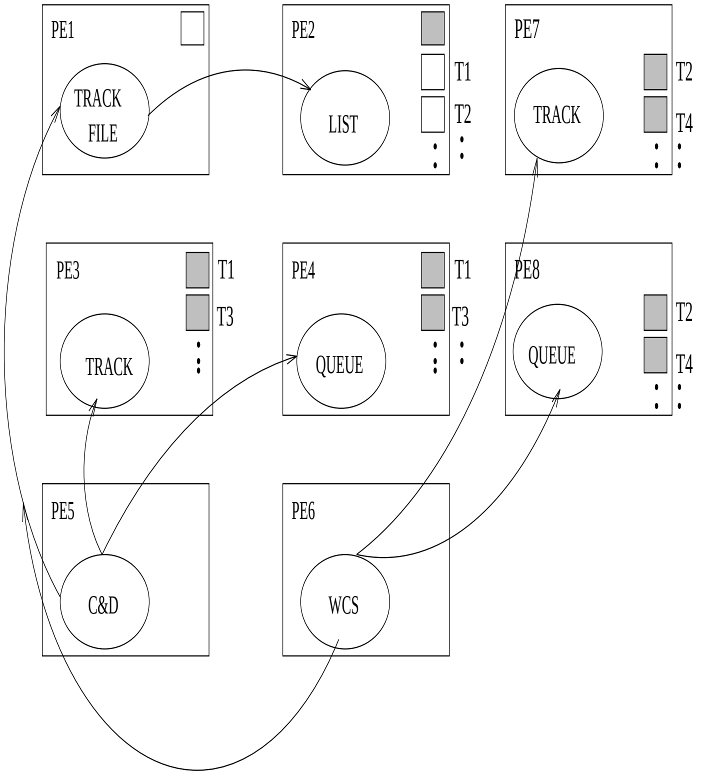


Figure 5: Assignment with cloning of track and queue instances.

module. Replication of the code module and redistribution of the track variables among the replicas can increase concurrency. Figure 5 shows the assignment following a single replication of the track and queue ADT instances. Note that the replication can increase performance by a factor of two: since the track file has been split in half, two tracks can be manipulated concurrently.

The explicit concurrency available in task-based systems, and the implicit concurrency available via ARPC can be exploited on modern hardware platforms, which are characterized by a large number of interconnected processing elements (PEs). For example, the Intel Paragon computer contains thousands of computing nodes, running according to the MIMD paradigm, and interconnected by a 2-dimensional mesh network. Its computing nodes contain multiple CPUs that share memory. Although there is shared memory within a node, there is no globally shared memory. Additionally, one CPU per node is dedicated to communication processing and the others are general-purpose processors.

## 2.3 The Reengineering of Two AEGIS Weapon System Modules

This Section presents summarizes two efforts [18] to reengineer portions of the AEGIS Weapon System from CMS-2 to Ada, and to migrate from a militarized AN/UYK-43 to commercial workstations. These projects were performed for two primary reasons: to aid in the refinement of a process for reengineering complex systems, and to provide proven algorithms for an experimental open system hardware and software environment (HiPer-D) directed at defining the future architecture and functionality of Navy ship computer systems.

The first reengineering experiment was completed in spring 1994, and the second experiment is scheduled to be completed in 1995. The reengineering experiments involved two different functional components from the AEGIS Weapon System Command and Control elements. The completed effort involved the Weapon Selection function, and the second effort focuses on the Surface Operations function. [1]

### 2.3.1 Weapon Selection Module

The first effort involved the reengineering of Weapon Selection, a module that automatically recommends a weapon to engage a target based on defined criteria. The reengineered Weapon Selection module is a rule based algorithm that evaluates the current tactical situation and automatically recommends when a target should be engaged. It also determines parametric values for such things as weapon intercept point and weapon intercept time. When required, this information is fed to the appropriate element to initiate engagement.

The goal of the first effort was to become familiar with the task of reengineering a set of mission critical AEGIS code. The requirements were that the code was to be translated and integrated into an open system infrastructure. In this effort, the software was translated from CMS-2 language into Ada using three different translators.

### 2.3.2 Surface Operations Module

The current effort is the reengineering of the Surface Operations module, which is responsible for making recommendations about steering a vessel to either reach or to avoid other vessels. this module makes recommendations about how an AEGIS ship should be maneuvered to adapt to certain situations. These include recommendations about the course to steer to obtain the closest point of approach to another vessel, the course to steer to try to avoid a torpedo, and the course to steer in order to remain a fixed distance from some vessel or shoreline.

The goal of this second effort is to validate a repeatable process that can be used to reengineer other portions of the AEGIS Weapon System, as well as to reengineer other computer-based systems. Thus, the emphasis is not on translation or evaluating translation, but is on identifying and performing the steps that are needed to produce a final product with high quality. At critical steps in this process, metrics are collected for analysis of the reengineering process and product.

## 3 The Reengineering Process

This section describes a process for reengineering which has evolved in conjunction with efforts to reengineer portions of the AEGIS Weapon system [18, 40]. The diagram [2] shown in Figure 6 indicates the major inputs and outputs of the reengineering process, which consist of the following items:

[1] AEGIS System software functions are composed of one or more units called modules. The functions that were chosen for reengineering are individual modules. In the case of the module from the Weapon Selection function, the module is one of many modules that make up the function. Whereas the module selected from the Surface Operations functions is actually the entire application.

[2] The diagram uses structured design notation, wherein circles denote processes, edges indicate the flow of data among processes, and boxes indicate entities that are external to the process.

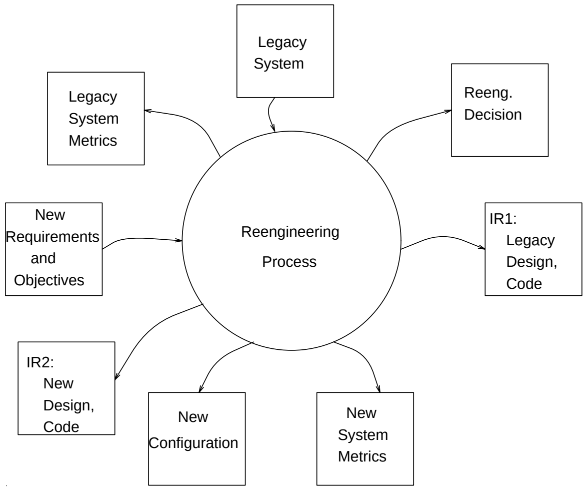


Figure 6: Context diagram for the reengineering process.

- Legacy system—the system (consisting of hardware, human and software elements) to be reengineered and all of its artifacts.
- IR1—an abstract representation of the legacy system, in machine-processable form.
- Legacy system metrics—concise characterizations of important aspects of the legacy system.
- Reengineering decision—the answer to the question "Which components from the legacy system should be reengineered?".
- New requirements and objectives—a description of the constraints and desirable properties that the reengineered system is to have.
- IR2—an abstract representation of the new system, in machine-processable form.
- New system metrics—concise characterizations of important aspects of the new system.
- New configuration—a description of the interactions of the hardware, operating system, application software and humanware of the new system.

As indicated in Figure 7, the first step of the reengineering process is reverse engineering, i.e., the capture of important features of the legacy system's hardware, software, and humanware. The reverse engineering process produces several outputs: IR1, legacy system metrics, and the reengineering decision. Given IR1, the legacy system metrics, and the new requirements and objectives, the task of software transformation manipulates IR1 until it satisfies the new goals and constraints. The transformation task is guided by the metrics for the legacy system. Transformation produces IR2 and metrics for the new software. Transformation is succeeded by configuration, which marries hardware, operating system, transformed software, and humanware. Software components are optimized for the execution paradigm provided by the hardware-operating system platform. The optimized software components are partitioned into tightly coupled clusters, which are assigned [30, 31, 43] onto the hardware platform in a way that (1) satisfies the new system requirements and (2) considers the new system objectives.

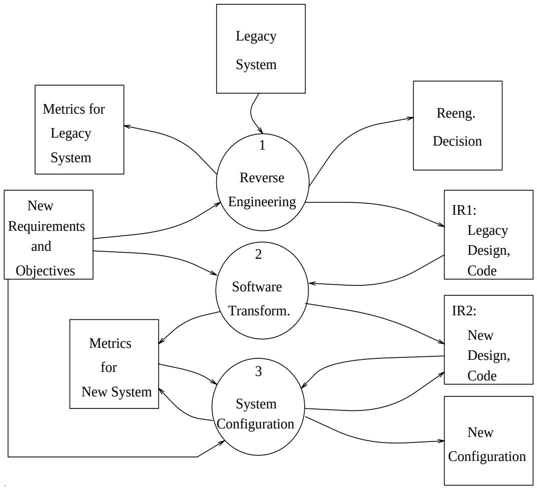


Figure 7: Steps of the automated reengineering process.

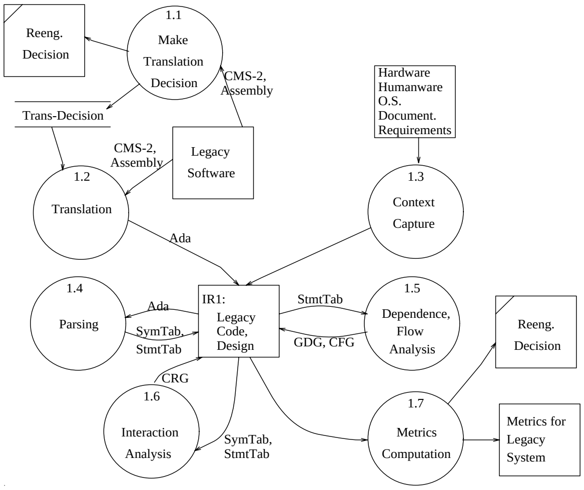


Figure 8: The reverse engineering process.

The output of the configuration process is a description of the partitioning, a specification of how partitions are assigned to processors, and a collection of metrics characterizing the new configuration. Following reengineering, an assessment of the reengineered product is made by comparing its metrics against the metrics for the pre-reengineered system. The three steps of the reengineering process are further explained in the remainder of this section.

## 3.1 Reverse Engineering

The goal of reverse engineering is to enable systems engineers to understand the important features of a legacy system's hardware, software, operating system, requirements, documentation and humanware. The approach taken in the reverse engineering of Navy systems such as AEGIS is indicated in Figure 8. The first step (process 1.1) is to make a decision about whether the legacy software, written in CMS-2 and assembly languages, should be translated into Ada. The decision is based on metrics extracted from CMS-2 tools such as OLTOOLS, and is also based on managerial and strategic factors (such as economics and technology advances) [21]. If the decision is "no", then the reengineering process terminates (such a decision with respect to translation implies an equivalent reengineering decision). Otherwise, the legacy software is translated into Ada and is incorporated into IR1. Additionally, the important aspects of the hardware, operating system (O.S.), humanware, documentation and requirements are captured in IR1. Following translation, the Ada code is parsed and the symbol table (SymTab) and the statement table (StmtTab) [13] are extracted and placed into IR1.

Process 1.5 of reverse engineering performs dependence analysis and flow analysis. Dependence analysis involves processing of the StmtTab to extract graphs that represent statement-level precedence relations due to control dependences, data dependences, and code dependences. Similarly, flow analysis extracts the statement-level control flow graph by examining the StmtTab. The general dependence graph (GDG) represents the precedence relationships and the control flow graph (CFG) contains the flow information [3].

During interaction analysis (process 1.6), interactions among tasks, packages, and procedures are identified. Specifically, interaction analysis produces a call-rendezvous graph (CRG), which is the union of the following two graphs:

- procedure call graph:
  - vertices represent program units such as subprograms, packages and tasks
  - directed edges represent caller-callee relationships (edges are directed from caller to callee)
- task rendezvous graph
  - depicts tasks, packages and subprograms as vertices
  - directed edges represent entities which rendezvous (edges are directed from the rendezvous initiator to the rendezvous acceptor)

## 3.2 Software Transformation

The translated code produced by the reverse engineering process has virtually the same design properties as the untranslated code. The only difference is in the implementation language, thus the post-translation product could hardly be called a "reengineered system." The tasks of transformation, optimization and configuration perform the core of the (re)engineering process. Following reverse engineering, the software design and code contained in IR1 are transformed to improve important properties, to meet requirements and to achieve objectives. Transformations are guided by the metrics described in [40]. For example, object-orientedness can be enhanced by considering (in IR1) the undesirable sources of low cohesion, high coupling, poor information hiding and low encapsulation. Similarly, concurrency and communication metrics can drive transformations to improve scalability of software by restructuring, to reduce intermodule communication and to enhance intermodule concurrency. Also, traditional compiler optimizations such as strength reduction, code motion and dead code elimination [1] can be performed, even by commercially available tools. During the transformation stage, components that were not reengineered, but were developed from scratch, or were reused from previous development

[3] Although this information involves fine-grain program units, it is needed to allow later phases of analysis to compute accurate metrics at higher levels of granularity.

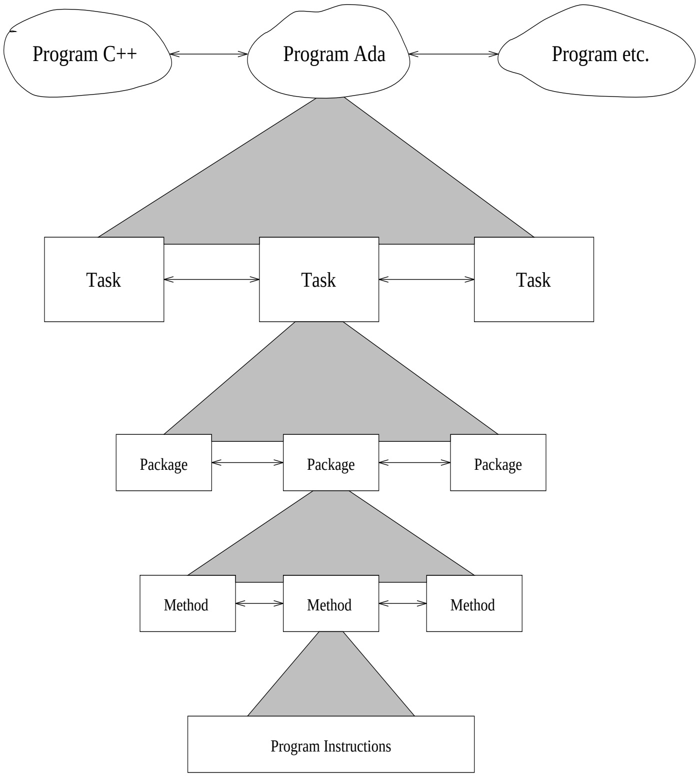


Figure 9: Mission critical software architecture.

efforts, are integrated with the transformed components. Given the complete set of program components, threads of control are identified and Ada tasks are inserted to define the threads. The timing properties of the system are also indicated with respect to the tasks. For example, it may be stated that a particular task has a deadline and a period, or that a particular task will execute each time that a certain event occurs. Due to the insertion of concurrent tasks, the transformation process must insert code to perform synchronization and deadlock handling. Furthermore, the fault tolerance properties (e.g., the degree of replication) of the software are specified with respect program units such as tasks, packages and subprograms.

### 3.3 System Configuration

Following software transformation, the software components (tasks, packages and procedures) are partitioned into tightly coupled groups (based on communication and concurrency relations among components), and each group is assigned [30, 43] to a processor of a parallel or distributed computer. Following this phase, distribution metrics are computed, and integration testing of program components is performed. The assignment of partitions to processors, and even the definition of partitions, continue to evolve during operation of the system, in order to adapt to changing load conditions and hardware faults.

## 4 Intermediate Representation

This section defines a language-independent intermediate representation (IR) for capturing computer-based systems' features that are essential for the reengineering process described in the previous section.

The systems under consideration have the structure depicted in Figure 9). We term this structure the *mission critical software architecture (MCSA)* [38], in order to contrast it with the *grand challenge software architecture* Such software systems are composed of several layers (or tiers). Elements at tier $i$ are implemented in terms of elements at tier $i-1$. Tier 1 consists of independently developed programs, which may be implemented in different languages. At tier 2 are tasks (independent threads of control), which may share resources, and are permitted to run concurrently. Tier 3 is composed of modules with multiple entry points (as in CMS-2), ADT packages (as in Ada, Modula, and Clu) and object classes (as in C++, Smalltalk and Eiffel). The employment of tier 3 components provides conceptual clarity, enables tier 2 tasks to be implemented "on top of" abstractions exported by modules, and promotes module reuse. The elements of tier 3 are implemented in terms of subprograms—the tier 4 elements. Methods are implemented as a collection of instructions (tier 5 elements). This section describes how the IR represents information at the tiers 2, 3, 4 and 5 of the mission critical software architecture.

## 4.1 Task Tier

The task tier is represented as a directed graph $TRG$ = (V,E), wherein:

1. a vertex v in V denotes a task object, f(v);
2. an edge (x, y) in E indicates that the code of task object f(x) initiates a rendezvous with an entry provided by task object f(y).

In addition to rendezvousing with other tasks, Ada tasks may call subprograms. Furthermore, subprograms may initiate rendezvouses with tasks. Such interactions must be considered during the system configuration process. Thus, the call-rendezvous graph ($CRG$) combines the nodes and vertices of $TRG$, $CGRAPH_P$, and $CGRAPH_S$, and inserts directed edges representing calls from tasks to subprograms and indicating rendzvous initiations from subprograms and packages to tasks. A sample $CRG$ is given in Figure 10.

## 4.2 Package/Class Instance Tier

At the package instance level, a directed graph is used to show call relationships among instances. A program is modeled by directed graph, $CGRAPH_P = (V, E)$, where:

1. a vertex $v \in V$ denotes package instance, $f(v)$;
2. an edge $(x, y) \in E$ indicates that the code of instance $f(x)$ calls some subprogram(s) provided by instance $f(y)$.

A sample $CGRAPH_P$ is given in Figure 3.

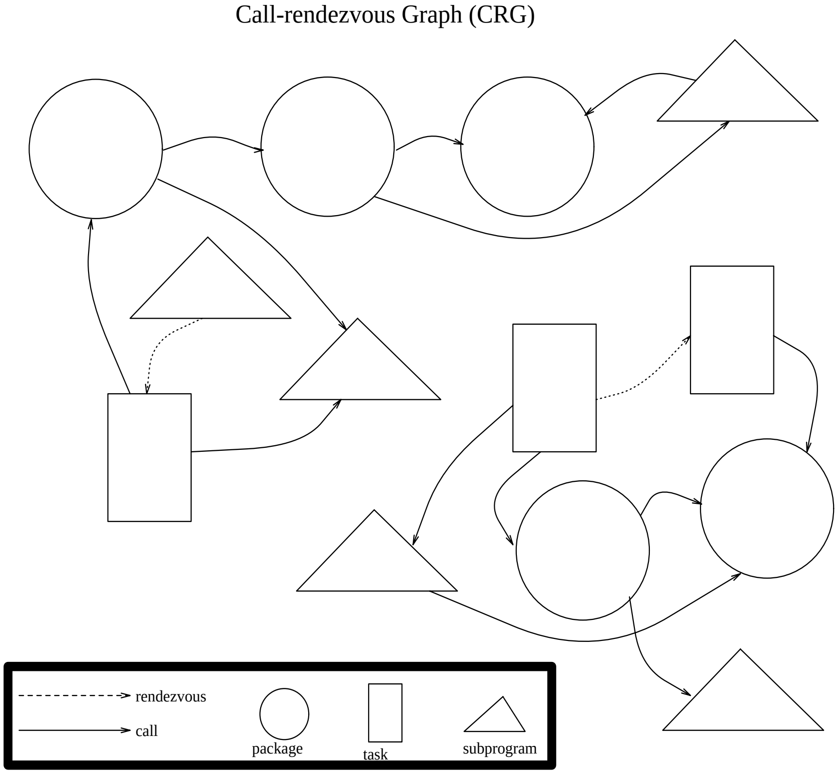


Figure 10: A call-rendezvous graph ($CRG$).

### 4.3 Subprogram Tier

At the granularity of the subprogram, a directed graph, $CGRAPH_S = (V, E)$, is used to represent the call relationships as follows:

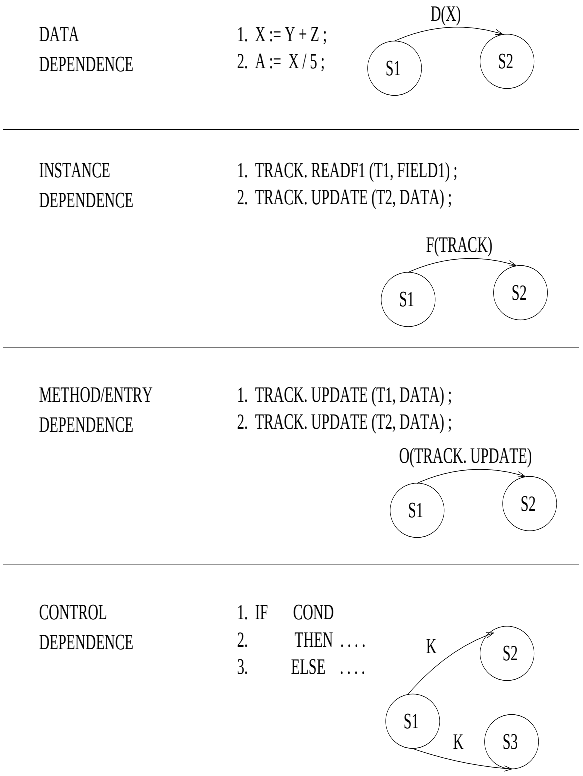
1. a vertex $m \in V$ denotes a subprogram $f(m)$
2. an edge $(m, n) \in E$ indicates that the code of subprogram $f(m)$ calls subprogram $f(n)$.

The edges of the graph are labeled to indicate call frequencies and metrics for communication and concurrency.

### 4.4 Statement/Instruction Tier

At the statement level (or at the assembly language instruction level), dependence graphs represent program statements as nodes and use directed edges to denote statement ordering implied by the dependences in a source program (see Figure 11). Different kinds of ordering requirements are represented in different dependence graphs, as described below.

- Data dependence graph (DDG): a directed edge denotes a data dependence (destination and source nodes need the same value).
- Control dependence graph (CDG): a directed edge denotes that flow of control to the destination statement is determined by the source statement.



```
1. X := Y + Z ;
2. A := X / 5 ;
```

```
1. TRACK. READF1 (T1, FIELD1) ;
2. TRACK. UPDATE (T2, DATA) ;
```

```
1. TRACK. UPDATE (T1, DATA) ;
2. TRACK. UPDATE (T2, DATA) ;
```

```
1. IF    COND
2.        THEN ....
3.       ELSE  ....
```

Figure 11: Statement dependence graphs.

- Instance dependence graph (IDG): an undirected edge denotes an instance dependence—two nodes use operations exported by the same instance [4]
- Subprogram dependence graph (SDG): an undirected edge denotes a subprogram dependence (two nodes use the same subprogram of a particular instance).

In addition to the dependence graphs, the control flow graph (CFG) is extracted at the statement level, indicating the sequential flow of control dictated by the order of the statements in the source code.

# 5 Reverse Engineering Tools for Constructing the Intermediate Representation

The reverse engineering efforts of the NJIT/NSWC team have resulted realizations of software tools to extract the IR from Ada programs. In this section, the techniques and algorithms embodied in the tools that construct the intermediate representation are presented.

[4] The IDG was invented for the purpose of ADT instance clone analysis.

## 5.1 Construction of Call and Task Rendezvous Graphs

Since the use relation among program units (tasks, packages, and procedure ) is explicitly declared in the source code, call graphs can be easily constructed during program parsing. The procedure for building call graphs is shown in Figure 25. The partial call graph of the package $Concatenation - Processor$ is shown in Figure 26. As mentioned before the CG is built during program parsing. During the parsing, information is collected from the Ada parser production rules. From this we build a symbol table for each program unit which contains, besides other relevant information, the call list for that unit. The complete application CG is an union of all the individual units' call lists.

Due to special language constructs that must be used in the source code, the task rendezvous graph, like the call graph, can also be easily constructed during program parsing. This process is similar to the call graph construction using the symbol table. The procedure for building task rendezvous graphs is also shown in Figure 25. The partial Task Rendezvous graph of a procedure $test1$ is shown in Figure 24.

Given the call graph and the task rendezvous graph, the IR we named call rendezvous graph is a simple union of the previous two as shown in Figure 25.

## 5.2 Construction of Statement-Level Graphs

In this section, a *statement table* is introduced first, which is used to collect information needed for constructing the statement-level intermediate representation of programs. This is followed by the algorithms for building the different dependence graphs. An Ada package *Concatenation_Processor* with a single method *Concatenate* (shown in Figure 12) is used to illustrate the dependence analysis techniques.

### 5.2.1 Statement Tables

There are many ways to construct program dependence graphs [8, 6, 3, 10]. For object-based systems, PDGs can be constructed at compile time, for each module, Define a *Statement Table* which contains the following attributes for each statement:

1. *Statement Type* indicates the type of the statement (e.g., *method call*, *if-then-else*, or *while loop*)

2. *Dependence Nesting Level* keeps track of the number of *region nodes* on the path from the root to it. (A *region node* is defined as a virtual node which has a zero execution time. A region node is used to group all the nodes that have a dependence relation on the same node, by forcing all those nodes to depend on the region node, and letting the region node depend on the single node.)

3. *Address* is a line number in the source code.

4. *ADT Instances Used* is the set of ADT instances directly used by the statement.

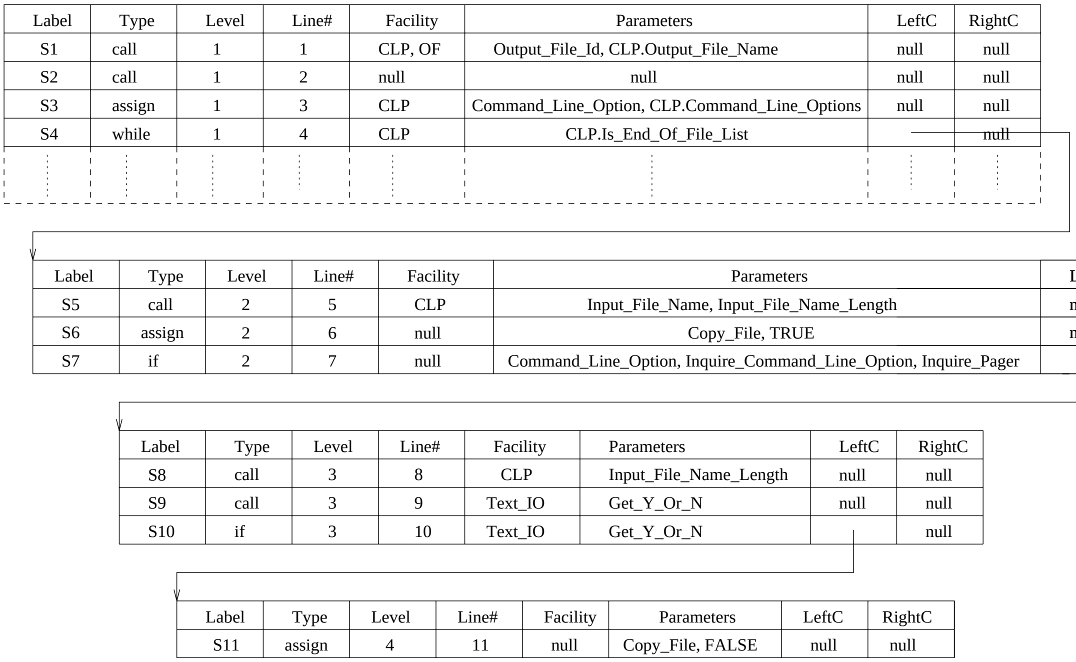
5. *Parameter List* is the set of variables used by the statement.

```
WITH Output-File, Input-File, Command-Line-Processor, Error-Log;
WITH Text-IO;
PACKAGE BODY Concatenation-Processor IS
-- Purpose
-- The Concatenation-Processor CATs the files together.
--
-- Initialization Exceptions (none)
-- Notes (none)

   USE Command-Line-Processor;
   PROCEDURE Concatenate IS
   -- Purpose
   -- Concatenate performs the Concatenation-Processor functions.
   --
      Output-File-Id : Output-File.File-Type;
      input-File-Id : Input-File.File-Type;
      Input-File-Name-Length, In-Line-Length : INTEGER;
      Input-File-Name, In-Line : STRING(1..400);
      Get-Y-Or-N : Character;
      Command-Line-Option : Command-Line-Processor.Command-Line-Options-Type;
      Copy-File : BOOLEAN;

   BEGIN
S1    Output-File.Create(Output-File-Id, Command-Line-Processor.Output-File-Name);
S2    Error-Log.Open("");
S3    Command-Line-Option := Command-Line-Processor.Command-Line-Options;
S4    WHILE NOT (Command-Line-Processor.Is-End-Of-File-List) LOOP
S5      Command-Line-Processor.Next-Input-File(Input-File-Name, Input-File-Name-Length);
S6      Copy-File := TRUE;
S7      IF (Command-Line-Option = Inquire) OR (Command-Line-Option = Inquire-Pager) THEN
S8        Text-IO.Put(”Copy File: ” & Input-File-Name(1..Input-File-Name-Length) & ” (Y/N) ?”);
S9        Text-IO.Get(Get-Y-Or-N);
S10       IF (Get-Y-Or-N /= ’Y’) AND (Get-Y-Or-N /= ’y’) THEN
S11         Copy-File := FALSE;
          END IF;
        END IF;
S12     IF Copy-File THEN
S13       Input-File.Open(Input-File-Id, Input-File-Name(1..Input-File-Name-Length));
S14       IF (Command-Line-Option = Pager) OR (Command-Line-Option = Inquire-Pager) THEN
S15         Output-File.Put-Line(Output-File-Id, ”–::::::::::::”);
S16         Output-File.Put-Line(Output-File-Id, ”–” & Input-File-Name(1..Input-File-Name-Length));
S17         Output-File.Put-Line(Output-File-Id, ”–::::::::::::”);
          END IF;
S18       Input-File.Get-Line(Input-File-Id, In-Line, In-Line-Length);
S19       WHILE NOT Input-File.End-Of-File(Input-File-Id) LOOP
S20         Output-File.Put-Line(Output-File-Id, In-Line(1..In-Line-Length));
S21         Input-File.Get-Line(Input-File-Id, In-Line, In-Line-Length);
          END LOOP;
S22       Input-File.Close(Input-File-Id);
        END IF;
      END LOOP;
S23   Output-File.Close(Output-File-Id);
   END Concatenate;
 END Concatenation-Processor;
```

Figure 12: An Ada Package Body

6. *Child* points to a statement table containing all its dependents in another dependence level. If a statement has more than one group of statements depending on it, a *Child* field is created for each of the groups. This occurs when the statement is an if-statement, case, or loop, where there exist multiple execution paths. For all statements which do not create multiple branches, the child field is just a *null* pointer.

For method *Concatenation_Processor* in Figure 12, the statement table of the method is shown in Figure 13. In the statement table, two child fields are created for each *while* or *if* statement.



| Label | Type | Level | Line# | Facility | Parameters | LeftC | RightC |
|---|---|---|---|---|---|---|---|
| S1 | call | 1 | 1 | CLP, OF | Output_File_Id, CLP.Output_File_Name | null | null |
| S2 | call | 1 | 2 | null | null | null | null |
| S3 | assign | 1 | 3 | CLP | Command_Line_Option, CLP.Command_Line_Options | null | null |
| S4 | while | 1 | 4 | CLP | CLP.Is_End_Of_File_List |  | null |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

| Label | Type | Level | Line# | Facility | Parameters | LeftC | RightC |
|---|---|---|---|---|---|---|---|
| S5 | call | 2 | 5 | CLP | Input_File_Name, Input_File_Name_Length | null | null |
| S6 | assign | 2 | 6 | null | Copy_File, TRUE | null | null |
| S7 | if | 2 | 7 | null | Command_Line_Option, Inquire_Command_Line_Option, Inquire_Pager |  | null |

| Label | Type | Level | Line# | Facility | Parameters | LeftC | RightC |
|---|---|---|---|---|---|---|---|
| S8 | call | 3 | 8 | CLP | Input_File_Name_Length | null | null |
| S9 | call | 3 | 9 | Text_IO | Get_Y_Or_N | null | null |
| S10 | if | 3 | 10 | Text_IO | Get_Y_Or_N |  | null |

| Label | Type | Level | Line# | Facility | Parameters | LeftC | RightC |
|---|---|---|---|---|---|---|---|
| S11 | assign | 4 | 11 | null | Copy_File, FALSE | null | null |

Figure 13: The Statement Table of method *Concatenate*

### 5.2.2 Construction of Control Dependence Graphs

A CDG of a method can be directly constructed from the statement table of the method since the statement table describes the control dependence relations among statements. A special region node called *entry* is added to the CDG and it indicates that all statements of the method are control dependent upon the *entry* node of the method. Also, for a statement which has two or more branches (like an if or a loop statement), a *region node* is added to the CDG for each branch. Thus the start of a branch is indicated by the region node and the region node then becomes control dependent upon the statement that branches. All statements in each branch are control dependent upon the region node.

The algorithm for building a CDG from a statement table is shown in Figure 14. The CDG shown in Figure 15 is constructed by using the statement table of Figure 13 as input to the algorithm.

```
BuildCDG(StaTab : StaTab_TYPE, entry : NODE_TYPE)
  var Q: QUEUE of node;
    x, y, z: NODE_TYPE;
  begin
    ENQUEUE(entry, Q):
    while not EMPTY(Q) do
      begin
        x := FRONT(Q);
        DEQUEUE(Q);
        for each none NULL ChildStaTab C of x in the StaTab do
        /* ChildStaTab is either x.LeftC or x.RightC */
          begin
            if (x.Type = "if") then
              begin
                y := getRegionNode; /* get a new region node */
                insert(x,y,CDG); /* insert an edge from x to y in the CDG */
              end
            else
              y:=x;
            for each entry N in C do
              begin
                z := getNode(N); /* get a new node with the label, N.label */
                insert(y,z,CDG);
                ENQUEUE(z,Q);
              end for
          end for
      end while
  end BuildCDG
```

Figure 14: Algorithm for building CDGs.

### 5.2.3 Construction of Data Dependence Graphs

Data dependence graphs (DDGs) describe the data dependence relationship among statements in a method or task. The DDG of a method can be constructed from the control flow graph of the method by examining the data dependence relations along the control flow of the statements of the method. The algorithm for building the DDG of a method is presented in Figure 16, Figure 17, and Figure 18. The DDG of the *Concatenate* method is shown in Figure 19.

### 5.2.4 Construction of Program Dependence Graphs

A *program dependence graph (PDG)* [8] of a method represents a union of the control dependence graph and the data dependence graph of the method. The algorithm for building the PDG of a method from its DDG and CDG is shown in Figure 20. Figure 21 (a) shows the PDG for *Concatenate*.

### 5.2.5 Construction of Instance Dependence and General Dependence Graphs

In PDGs, all statements that are immediate successors of a common region node can run concurrently. In general, any two statements can run concurrently if they have neither direct nor transitive precedence relations. Note the two conclusions above may not hold when instance dependence is considered. For example, if two calls go to the same ADT instance, they cannot run concurrently even though they have neither direct nor transitive precedence relations, since the code of an ADT instance cannot be used concurrently (though it may be reentrant) since only one PE contains a copy of the code.

By adding instance dependence into the PDG of a method or task, a graph representing three kinds of dependences - control, data, and instance is created. The new graph is called a **General Dependence Graph (GDG)** of the method or task. The GDG of the method *Concatenate*

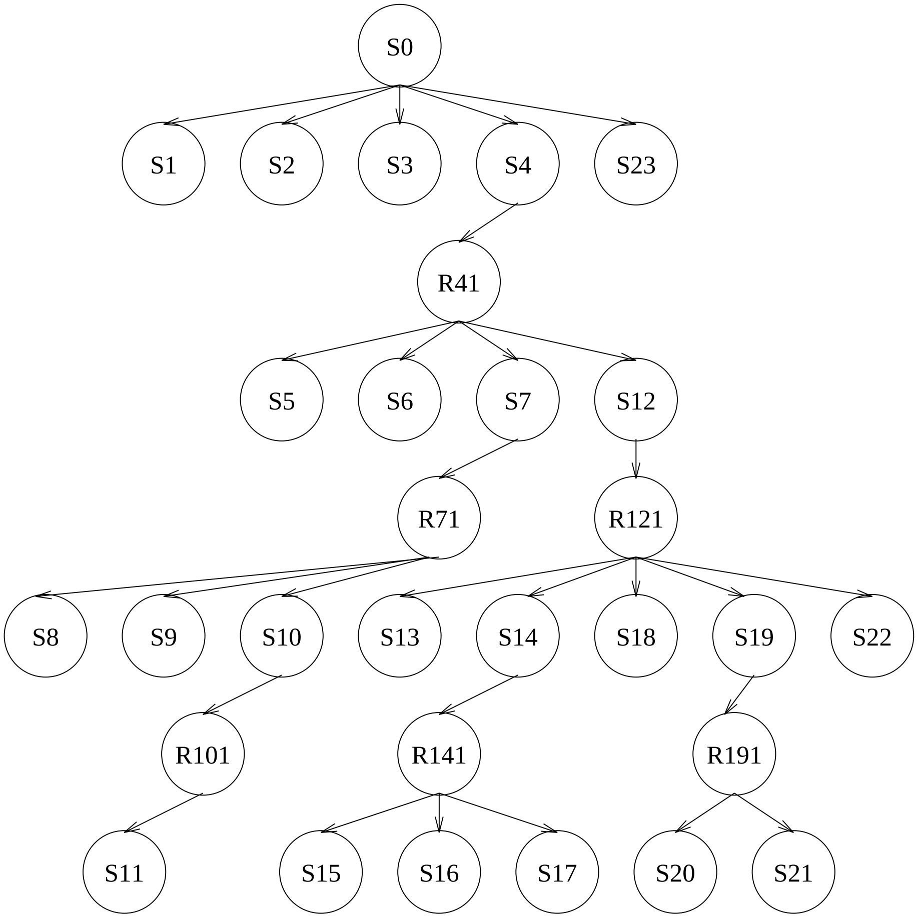


Figure 15: The CDG of method *Concatenate*

is shown in Figure 22. Since local statements (not incurring method calls) have to be executed sequentially, successive local statements can be treated as a super node called *local super node*. With super nodes, the GDG of a method can be simplified. Figure 21 (b) shows the simplified GDG of method *Concatenate*. For each used ADT instance of a method, the statements using the ADT instance form a subgraph called an *instance dependence graph (IDG)*. The number of subgraphs of a method's GDG is equal to number of ADT instances used by the method. The IDGs of method *Concatenate* are shown in Figure 23.

An instance dependence edge (in the IDG) represents contention for the code of an ADT instance. This instance dependence edge could be removed by cloning the code, thus enhancing concurrency. This is not true if two nodes using the same ADT instance have date or control dependence. In other words, the data or control dependence prevents the two statements from executing concurrently. Detailed discussion on ineffective instance dependence relations can be found in [42]. Therefore, while building the GDG, we add only those instance dependences into the PDG that connect a node with one of its siblings. In other words, we do not add any instance dependence that connects a node to its descendant in the PDG.

```
SearchDD(tt : StatementType)
  PS : stack(StatementType)
  begin
    if (tt.rightc $\neq$ null) or (tt.leftc $\neq$ null) then
      Push tt.rightc & tt.leftc into stack PS;
    else
      begin
        st = successiveStatement(tt);
        if (st $\neq$ null) then stack.push(st, PS);
      end
    while not stack.empty(PS) do
      begin
        st = stack.pop(PS);
        if (st $\neq$ null) then
          begin
            if (checkDD(st,tt) = true) then
              begin
                DDG(tt,st) = true;
                Remove (st.Parameters $\cap$ tt.Parameters) from tt;
                if no more parameters in tt remain to be checked then
                  while not stack.empty(PS) do st = stack.pop(PS);
                else
                  begin
                    st = successiveStatement(tt);
                    if (st $\neq$ null) then stack.push(st, PS);
                    else flag = true;
                  end
              end
            else
              if (st.rightc $\neq$ null) or (tt.leftc $\neq$ null) then
                Push st.rightc & st.leftc into stack PS;
              else
                begin
                  st = successiveStatement(tt);
                  if (st $\neq$ null) then stack.push(st, PS);
                end
          end
        if (flag = true) then
          while not stack.empty(PS) do st = stack.pop(PS);
      end while
  end SearchDD
```

Figure 16: Algorithm for building DDG (searching for data dependence)

# 6 Navigating Intermediate Representations via Hypertext

An essential prerequisite for reengineering consists in an understanding of the original program across different levels of abstraction. The information provided through the intermediate representation presented in the previous section greatly helps towards such an understanding, but it probably is most helpful only after the basic workings of the programs have been comprehended. The reason for this is that the intermediate representation captures relevant features of entities on a certain tier, e.g. the statement/instruction tier, which are closely related, but represents these features in separate graphs, and displays them separately to the user. A similar observation holds for information captured on different tiers. This section describes the use of some hypertext techniques used to retain the contextual and pragmatic relationships among the different pieces of information in the intermediate representation.

## 6.1 Zooming

When trying to obtain an understanding of a program, it is often necessary to switch between different levels of abstraction. One might, for example, start at a high level, looking at the main components the program consists of, and the way they are integrated. Then, in order to understand the role of one of the components more clearly, a more detailed look at it can be necessary, if possible without losing the information displayed previously. Such techniques have been widely applied in various contexts, and are often referred to as 'zooming'. In our context

```
successiveStatement(st : StatementType) returns StatementType;
  begin
    if (st.rightc $\neq$ null) or (st.leftc $\neq$ null) then
      begin
        if (st.leftc $\neq$ null) then
          return (st.leftc);
        if (st.rightc $\neq$ null) then
            return (st.rightc);
      end
    else
      if (st.sibling $\neq$ null) then
        return (st.sibling);
      else
        begin
          while (st.parent $\neq$ null and st.parent.sibling = null) do
            st := st.parent;
          if (st.parent $\neq$ null and st.parent.sibling $\neq$ null) then
            return (st.parent.sibling);
          else
            return null;
        end
  end successiveStatement
```

Figure 17: Algorithm for building DDG (searching for the successive statement)

```
checkDD(st, tt : StatementType) : boolean;
  begin
    if (st.Parameters $\cap$ tt.Parameters = $\emptyset$) then
      return true;
    else
      return false;
  end checkDD
```

Figure 18: Algorithm for building DDG (checking data dependence among parameters)

here, it can be realized in several ways, depending on the way the initial information is displayed.

A starting point could be the directed graph displaying the call relationships among instances at the package/class level. Then one of the packages or classes can be inspected more closely, by opening another window displaying its contents. In order to keep track of the hierarchical relationships between windows, it is useful to indicate this, e.g. by drawing thin lines between the detailed and the short view of the package. As an example, consider a system similar to the one depicted in Figure 3, but with a larger number of modules. In the high-level view, the internals of the single modules would not be visible, and only displayed on demand.

On the other hand, if a textual representation is used initially, similar approaches can be applied to display different levels of detail in the source code. Whereas the graph representation on the package level would provide information on the relationships among the packages, the corresponding textual representation describes the interfaces of the packages, possibly augmented by comments. *Folding editors* or *outline modes* make uses of these techniques, too, and can be adapted for our purpose with moderate effort. Figure 27 shows the multiple zooming layers.

## 6.2 Cross-Linking

So far, textual and graphical representation of the components have been considered separately. Tying these two together can provide further enhancement to the understanding of the program. One method here is to apply certain operations like highlighting to both representations simultaneously, thus emphasizing the correspondence of particular elements in the different representations. Going one step further, the inspection of one component in one representation can

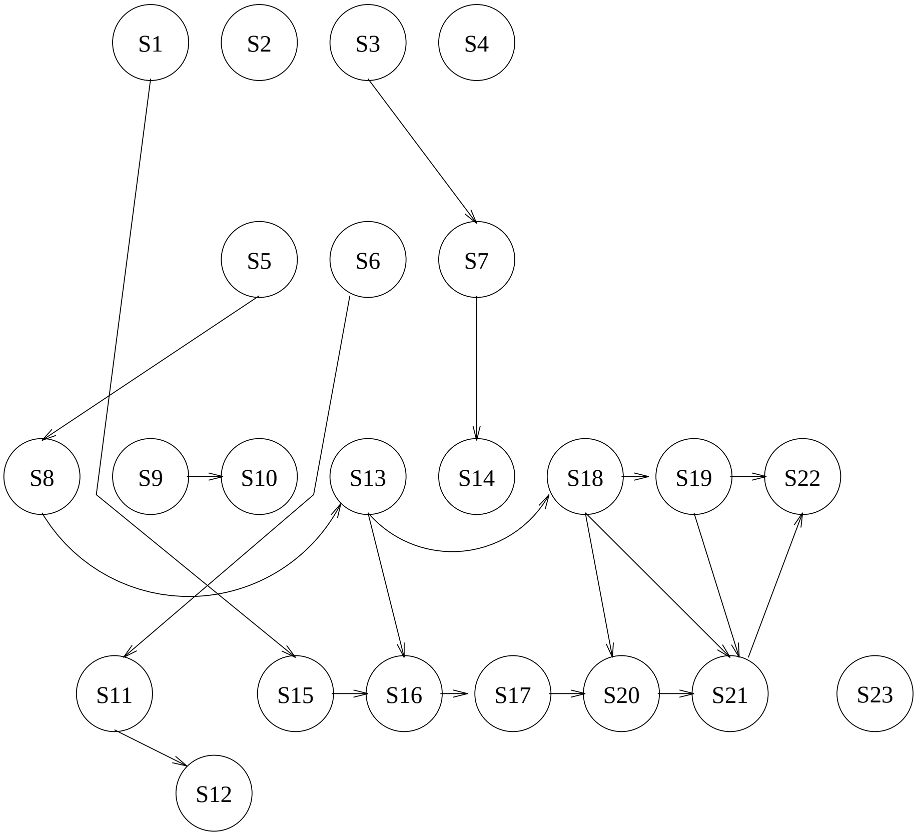


Figure 19: The DDG of method *Concatenate*

trigger the display of that component in the other representation as well. Figure 28 sketches the integration between the graph and the text representation.

Obviously the integration of different views can be applied on various levels of abstraction. On lower levels, for example the statement/instruction tier, the picture becomes more complicated since different kinds of dependence graphs show different kinds of relationships between statements. Here the help provided by a method like highlighting corresponding statements in different views becomes even more meaningful. On the same level in the textual mode, hypertext can be used to provide within the source code the information residing in the edges of the respective dependence graph. Note, however, that in this case the information is not visually

```
BuildPDG(DDG,CDG,CDDG)
  begin
    copy CDG to CDDG;
    for each $S_i \rightarrow^d S_j$ in DDG do
      begin
        if $S_i$ is not the ancestor of $S_j$ in CDG then
          begin
            if parent($S_j$) is a region node which is the ancestor of $S_i$ in CDG then
              remove the edge from parent($S_j$) to $S_j$ in CDDG;
            add an edge from $S_i$ to $S_j$ in CDDG;
          end
      end for
  end BuildPDG
```

Figure 20: Algorithm for building PDGs

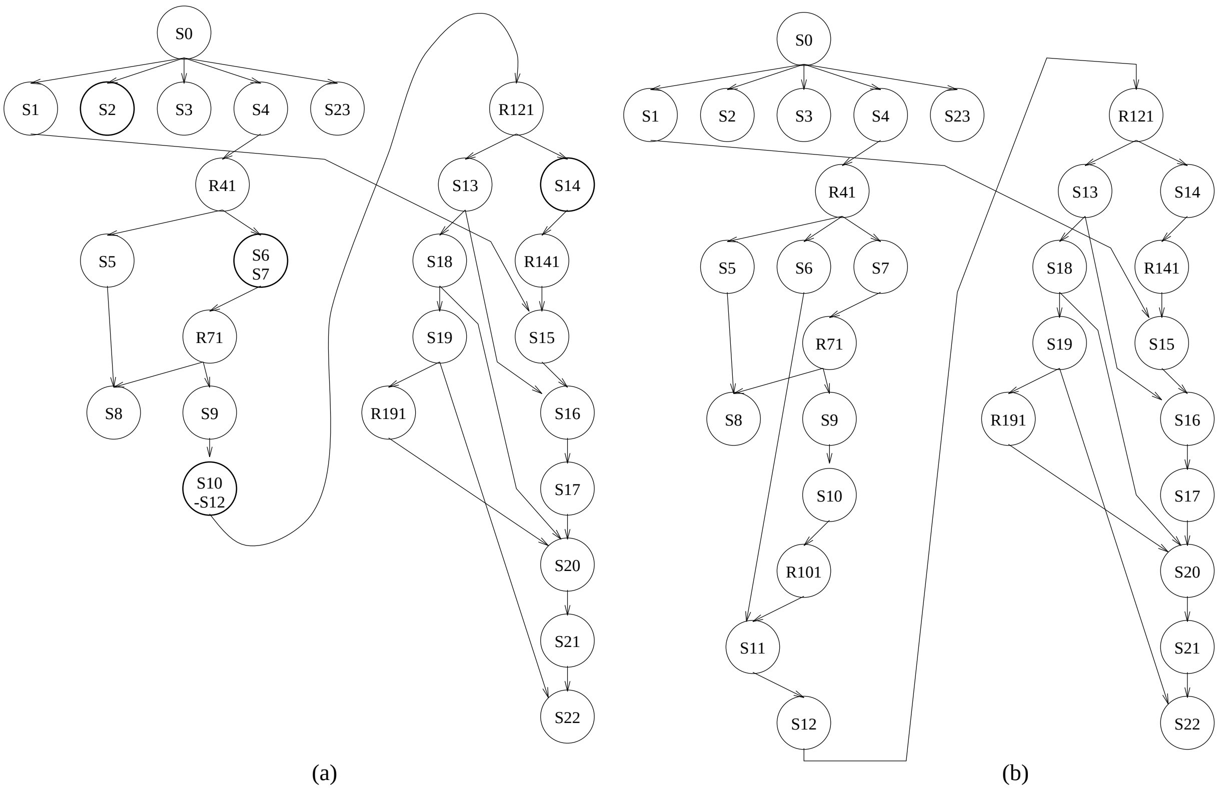


Figure 21: (a) The PDG of method *Concatenate* , (b) The simplified PDG of method *Concatenate*

displayed; it can only be exploited by following the respective link to where it points. Care must also be taken to distinguish the different dependences that may exist for one statement; this can be done via color-coding, or pop-up menus for selecting the desired dependence. Following such a link can be done in two ways: the focus of the current view changes, and the position in the source code to which the link points will be the new focus. This method has the disadvantage that the original context is lost, and after following a few links, one can get disoriented and have no idea in which part of the program one ended up. An alternative is to open a new window each time a link is being followed. The problem with this method is a potential proliferation of windows; also, most hypertext systems do not display the the connection between the two pieces of text belonging to a link.

### 6.3 Coordinating Views of Dependence Graphs

Especially on the lower tiers of the software architecture, the wealth of information provided in the intermediate representation can be very helpful for understanding the intricacies of the program. On the other hand, however, it can also present a major challenge to the person trying to understand what is going on in that program. One reason for this is that there are separate graphs displaying the different dependences, but all the graphs refer to one and the same program. Some particular elements will be present in different graphs, possible in different arrangements, and thus maybe difficult to identify. In the following we will describe some techniques which help the user to maintain his focus of attention throughout the inspection of one particular part of the program, while possibly viewing different types of dependence graphs.

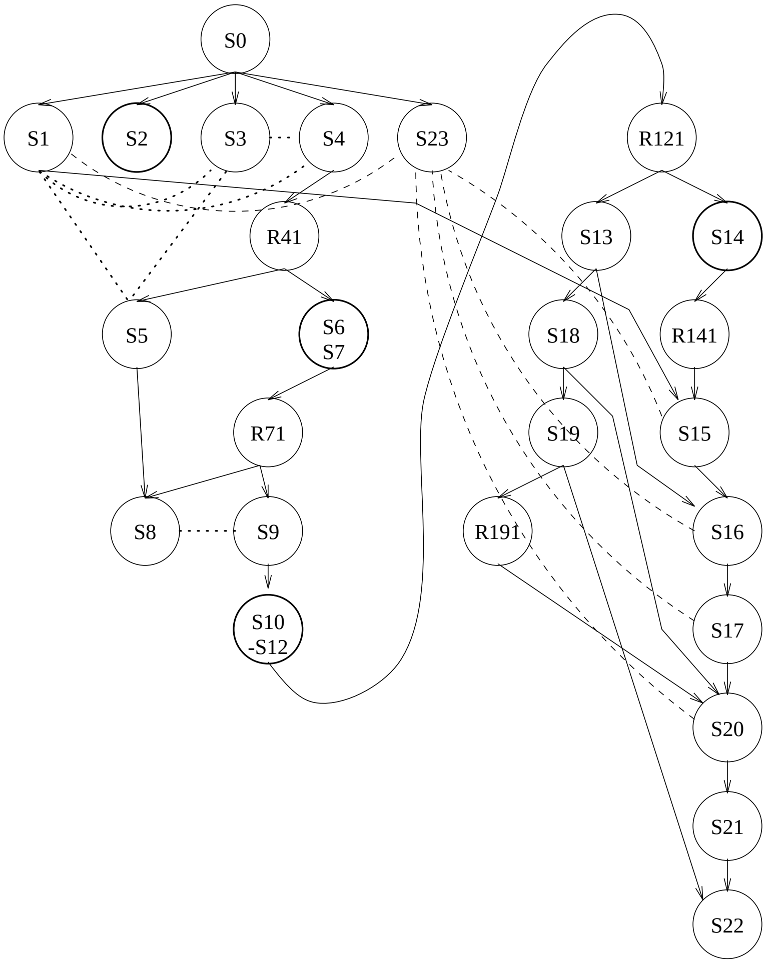


Figure 22: The GDG of method *Concatenate*

**Cross-Linking** The method of highlighting across different views mentioned before can offer some relief here for locating single elements; it is more difficult, however, to investigate the different dependences of a set of elements. In this case, one possibility is to extend highlighting by using different colors or other visual clues. In this approach, the different graphs are still displayed seperately; the system essentially helps the user with the identification of specific elements in separate graphs.

**Uniform View** Since all the dependence graphs are based on the same program, the nodes of the different graphs will be – possibly different – subsets of the overall set of nodes given by the elements of the program on that particular level. One possibility now is to use the superset of all nodes as basis for the graphical display, and provide the user with a choice of link types corresponding to the different dependences. This would also allow the simultaneous display of several dependences, using colors or line thickness to distinguish between the dependences. The advantage of this method is the unchanged spatial arrangement of the nodes, making it easier for the user to stay oriented when comparing different dependences. On the other hand, this might lead to distorted arrangements for particular dependences, making it difficult to understand the relationships between the nodes for that type of dependence. In addition, the sheer number of

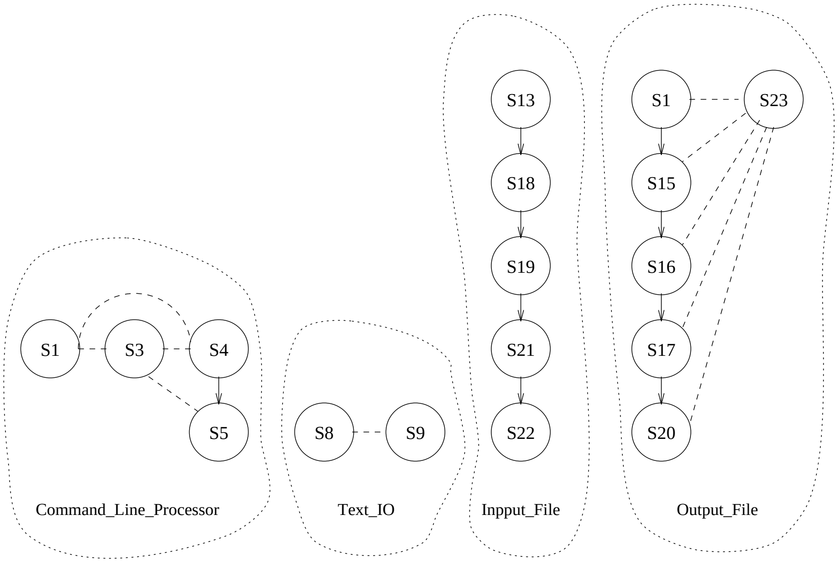
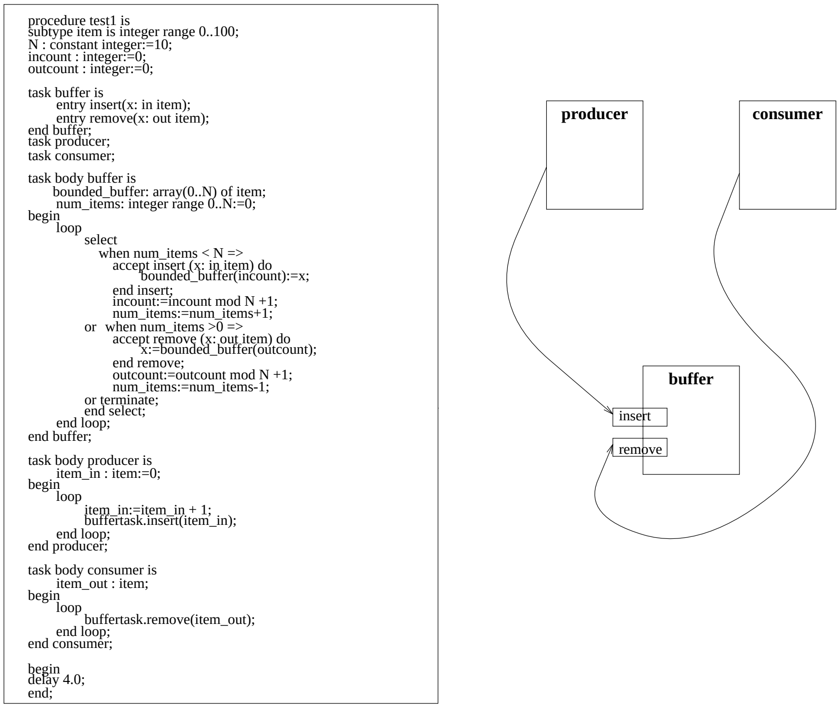


Figure 23: The IDGs of method *Concatenate*

nodes can present a problem for the visual representation: for a given type of dependence, the number of relevant nodes can be substantially smaller than the number of all nodes, and thus far easier to display.

**Merging** In many cases, different dependence graphs of one program will be rather similar As an example, consider the different dependence graphs for the method *Concatenate* shown in Figures 15-22. As shown in Section 5, the control, data and instance graph can be integrated into one single graph, the general dependence graph shown in Figure 22. This graph provides a lot of information about different aspects of the program, and indicates interactions across different types of dependences. If one is interested in one particular aspect of the program, e.g. the potential for concurrency which crucially depends on instance dependences, a different spatial arrangement of the nodes, however, can be much more helpful (see Figure 23).

**Overlays** Another way of inspecting different dependence aspects of a program is to use one dependence graph as starting point, and to "blend in" the edges and/or nodes representing another type of dependence. On the statement level, one can start from the control flow graph, and then add the additional information provided by the data dependence graph, for example. Note that with this approach it can be necessary to add nodes as well as edges, depending on the initial graph. This way one can familiarize oneself with one aspect of the program, and add another one without having to regain orientation because the arrangement of the nodes might have changed. In many cases, the resulting display can be similar to the one obtained from merging similar nodes as described in the previous paragraph. The advantage here is that the user can start from one graph and incrementally add information as needed, whereas in the merged graph everything is presented at once, and it may require considerable effort to separate different aspects. Whereas to the user, the dynamic aspect of adding information can be beneficial, it poses sopme challenges on the realization of the graph display: Graph matching needs to be done on the fly. Again, in many cases the various dependence graphs will be similar anyway; and for very dissimilar graphs, it is questionable if the overlay method is beneficial to



```
procedure test1 is
subtype item is integer range 0..100;
N : constant integer:=10;
incount : integer:=0;
outcount : integer:=0;

task buffer is
        entry insert(x: in item);
        entry remove(x: out item);
end buffer;
task producer;
task consumer;

task body buffer is
       bounded_buffer: array(0..N) of item;
        num_items: integer range 0..N:=0;
begin
        loop
                select
                    when num_items < N =>
                        accept insert (x: in item) do
                                bounded_buffer(incount):=x;
                        end insert;
                        incount:=incount mod N +1;
                        num_items:=num_items+1;
                or  when num_items >0 =>
                        accept remove (x: out item) do
                                x:=bounded_buffer(outcount);
                        end remove;
                        outcount:=outcount mod N +1;
                        num_items:=num_items-1;
                or terminate;
                end select;
        end loop;
end buffer;

task body producer is
        item_in : item:=0;
begin
        loop
                item_in:=item_in + 1;
                buffertask.insert(item_in);
        end loop;
end producer;

task body consumer is
        item_out : item;
begin
        loop
                buffertask.remove(item_out);
        end loop;
end consumer;

begin
delay 4.0;
end;
```

Figure 24: The (partial) task rendezvous graph of the procedure test1

the user.

The goal of this section was to illustrate the use of hypertext techniques which can provide some help for understanding a program to be reengineered. Most of them are directed at integrating the information captured by an analysis of various aspects of that program which are represented in the different dependence graphs.

# 7 Concurrency Metrics

Reverse engineering involves more than just capturing the IR—it involves analysis to identify potential concurrency. During the computer-based systems reengineering process, it is necessary to identify potential concurrency during reverse engineering. The concurrency information is used to guide the reengineering processes of software transformation and system configuration, which seek to produce a system with a high degree of concurrency. The metrics are also used to assess the concurrency in a reengineered system. Specifically, the following metrics are employed:

- Inherently sequential percentage [39] of methods, class/package instances, and tasks measured in various units: (1) the percentage of statements that can not execute concurrently, or (2) the percentage of ARPCs that can not execute concurrently.
- Inherently parallel percentage [39] of methods, class/package instances, and tasks defined as: (1) the number of statements that can execute concurrently, (2) the number of ARPCs that can execute concurrently, or (3) percentage of statements in the largest group.

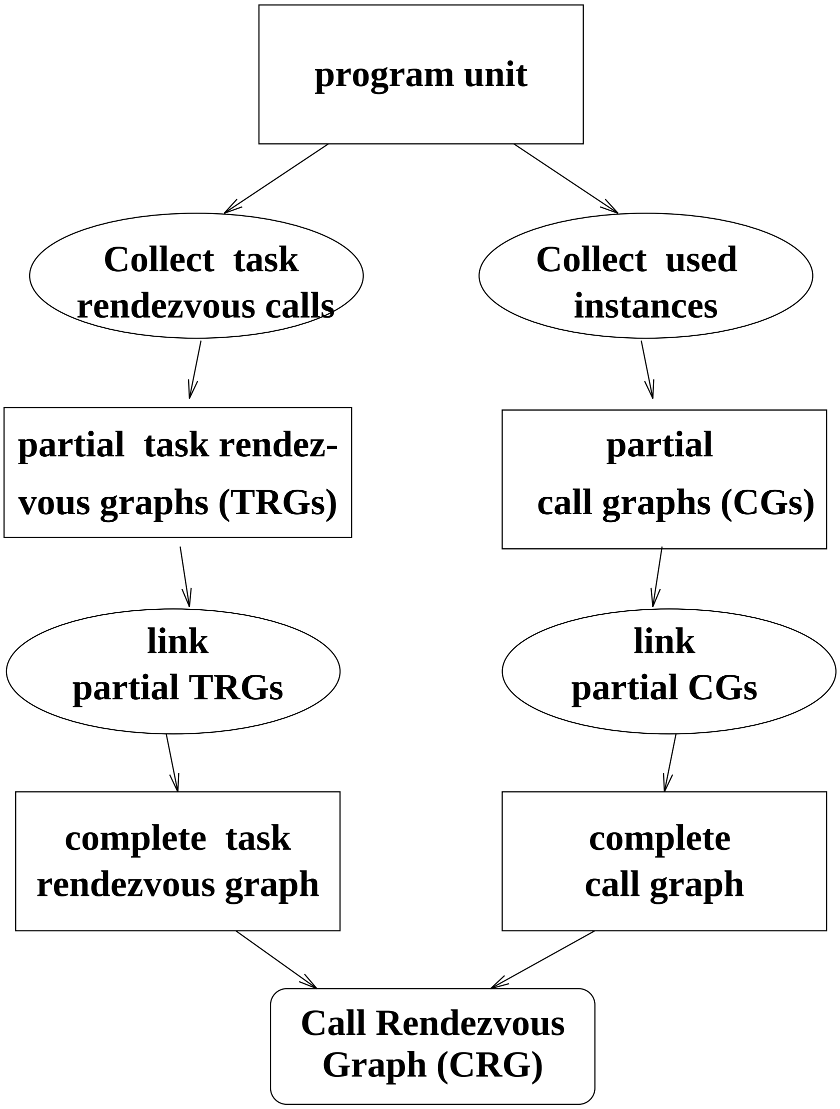
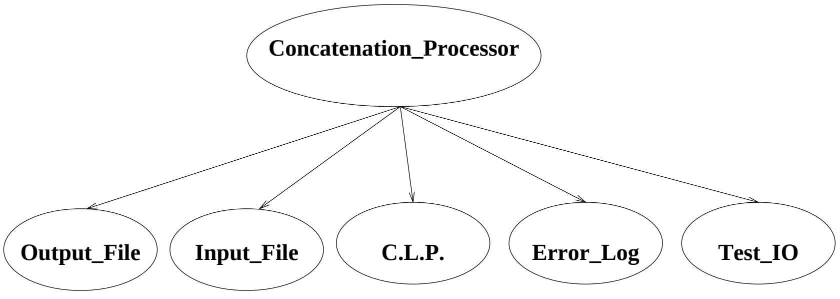

program unit
Collect task rendezvous calls
Collect used instances
partial task rendezvous graphs (TRGs)
partial call graphs (CGs)
link partial TRGs
link partial CGs
complete task rendezvous graph
complete call graph
Call Rendezvous Graph (CRG)

Figure 25: The procedures for building call graphs

- Concurrency dependences [32] (e.g., A and B can run concurrently iff B and C can run concurrently).
- Maximum number of replicas of methods and class/package instances that can be used concurrently [42, 36].
- The set of potentially concurrent entities, at each level of granularity (statement/instruction, method, class/package instance, task, and program) [32, 42, 39, 30].
- Potential concurrency among beads and tasks [39, 30].
- Amount (lines of code) of potential concurrency among methods, among class/package instances, among tasks, and among programs.

Our approach to obtaining concurrency metrics is *aggregation* of metrics across tiers of the mission critical software architecture (for example, see [32, 31, 29, 36, 25, 34]). Concurrency



Note: C.L.P. --- Command_Line_Processor

Figure 26: The (partial) call graph of the package Concatenation-Processor

metrics for a component at tier $i$ are obtained by aggregating concurrency metrics of components at tier $i-1$. For example, the concurrency metrics pertaining to a class/package instance are defined in terms of the concurrency metrics defined for the methods within the instance. This approach accommodates the layered structure of computer-based systems, and reduces the complexity of obtaining metrics for a component by abstracting information across levels.

# 8 Related Work

While much work has been published on the topics of reverse engineering and reengineering, little has been published within the context of concurrency enhancement in computer-based systems. In [4], an approach is presented for capturing abstractions inherent in software systems and for transforming those abstractions into an object-oriented paradigm; the focus was not on concurrency, but large-scale systems were considered. The consideration of concurrency is proposed in [12], by considering the translation of Unix systems calls into Ada constructs. Techniques and tools have been developed for source-to-source translation of program code [17, 2]; these tools are pragmatic, allowing a reengineered system to become operational quickly, but they do not attempt transformation of the nature required in computer-based Navy systems. Additionally, several techniques and tools have been developed to perform basic dependence analysis, including the Xinotech program composer [41], a tool and language independent IR developed by MITRE [16], and Refine [14], which performs reverse engineering of code written in Fortran, Cobol, C and Ada. However, none of these tools attempts to perform the analysis required for concurrency enhancement. Other techniques and tools for dependence analysis are presented in [7, 15, 5]. A hierarchical approach to reverse engineering was taken in [9], but the levels of the hierarchy were not based on granularity, as in our model, but consisted of implementation, structure, function and domain levels.

# 9 Conclusions

This paper describes a comprehensive process for the reengineering of computer-based systems. It considers the entire system, not just software. Indeed, it considers the interactions of software, hardware and humanware. The robustness of the process is seen by noting that it encompasses

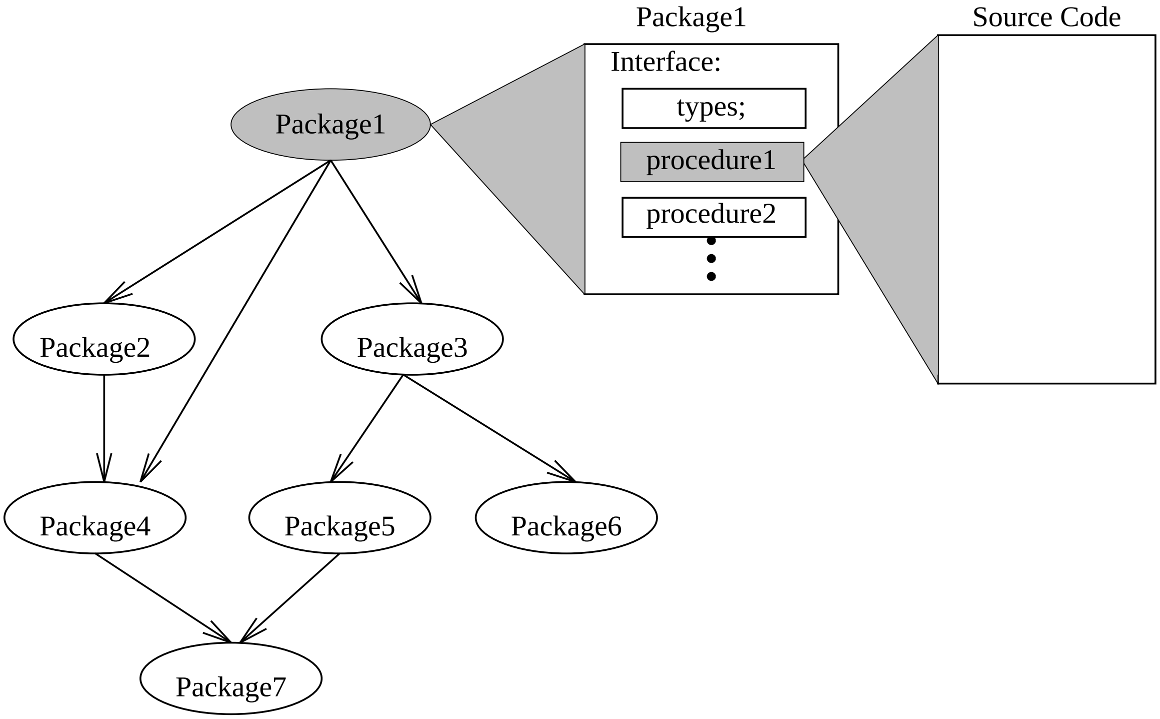


Figure 27: Zooming to get detailed views.

all major phases necessary for deploying a reengineered legacy system, not just the phases of reverse engineering and translation of software. A major strength of the process is that it has been, and continues to be, applied to the AEGIS Weapon System, a large, computer-based system of the United States Navy.

The focus of the paper is reverse engineering within the context of reengineering for concurrency and quality enhancement in computer-based Navy systems. Thus, computer-based Navy systems of the past and future are discussed, and two reengineering case studies are presented. Sections 4 and 5 define a novel view of software systems—a paradigm-independent IR based on the mission critical software architecture. Furthermore, a reengineering analysis toolset for constructing the IR at several tiers of the software architecture is presented. Building on the IR, the paper presents a metrics-based approach to reverse engineering and defines concurrency metrics at several levels of the software architecture. The concurrency metrics are required inputs into the reengineering phases of software transformation and system configuration.

Another accomplishment of the reengineering project is validation of the reengineering process. This is accomplished by the successful reengineering of the Weapon Selection and Surface Operations modules of the AEGIS Weapon System, and by the deployment of the reengineered modules within the Navy's HiPer-D distributed computing environment.

Ongoing work includes the application of the reengineering process to increasingly complex portions of the AEGIS Weapon System, application of the process to other computer-based systems, and automation of the process. Additionally, the metrics computation tool is being improved and refined, with the goal of making it available as a commercial product. Another important problem being addressed is the partitioning of program components and the assignment of partitions onto the processors of parallel and distributed systems. Techniques for software

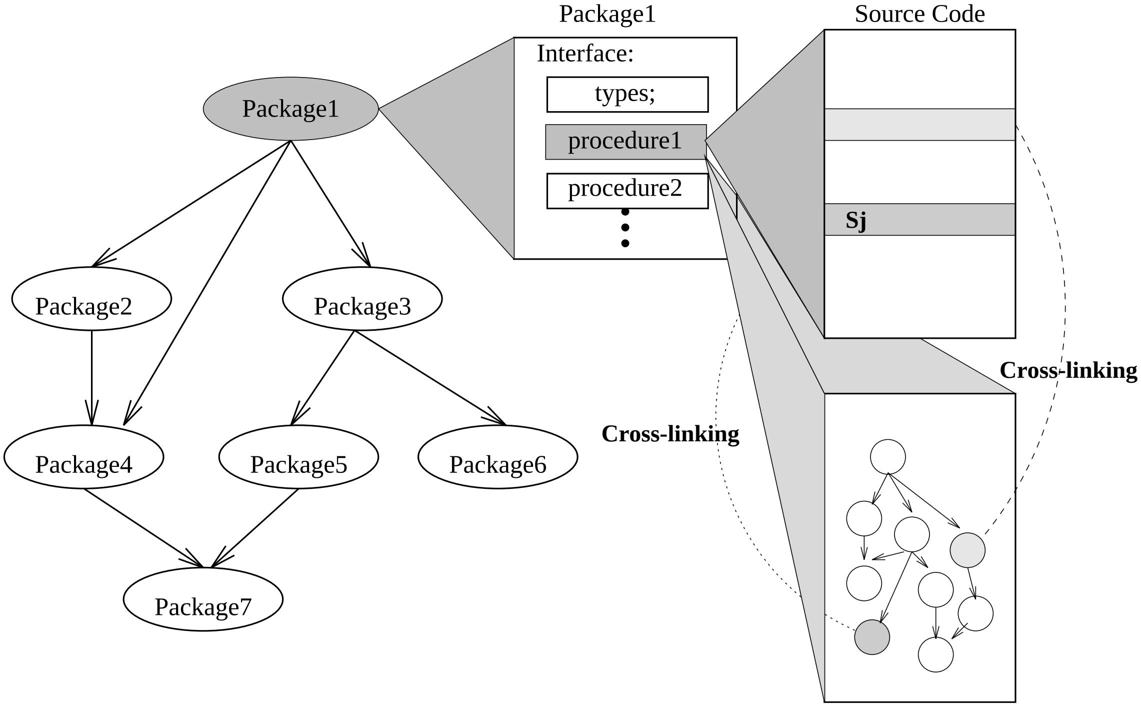


Figure 28: Cross-Link the same object in different views.

transformation are also being developed. To assist systems engineers with reengineering tasks that cannot be entirely automated, graphical techniques and tools for interactive transformation, partitioning and assignment are also being explored.

# References

[1] A. V. Aho, R. Sethi and J. D. Ullman, "Compilers: principles, techniques and tools," Addison Wesley, 1986.

[2] G. Arango et al., "Maintenance and Porting of Software by Design Recovery," *Proceedings of The Conference on Software Maintenance*, pages 42-49, IEEE CS Press, 1985.

[3] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein. The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-Driven Interpretation of Imperative Languages. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 257–271. ACM, June 1990.

[4] T. J. Biggerstaff, "Design Recovery for Maintenance and Reuse," *IEEE Computer*, volume 22, number 7, July, 1989.

[5] C. Castells-Schofield, "Engineering a Language-Independent Approach to Parsing for Analysis and Testing," *Vitro Tech.Journal*, volume 8, number 1, 1990.

[6] R. Cytron, J. Ferrante, B. K. Rosen, and M. N. Wegman. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. on Programming Languages and Systems*, 13(4):451–490, October 1991.

[7] S. Dietrich and F. Calliss, "A Conceptual Design for a Code Analysis Knowledge Base," *Software Maintenance: Research and Practice*, volume 4, 1992.

[8] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Trans. on Programming Languages and Systems*, 9(3):319–349, July 1987.

[9] M. Harandi and J. Ning, "Knowledge-Based Program Analysis," *IEEE Software*, volume 7, number 1, 1990.

[10] M. J. Harrold, B. A. Malloy, and G. Rothermel. Efficient construction of program dependence graphs. Technical Report 92-128, Clemson University, December 1992.

[11] T. J. Marlowe, A. D. Stoyenko, S. P. Masticola, and L. R. Welch, "Schedulability-Analyzable Exception Handling for Responsive Languages," *Journal of Real-Time Systems*, to appear.

[12] N. Prywes, G. Ingargiola, I. Lee and M. Lee, "Reengineering Concurrent Software into Ada," *Proceedings of The Fourth Systems Reengineering Technology Workshop*, pages 157-177, Naval Surface Warfare Center, February 1994.

[13] B. Ravindran, "Extracting parallelism at compile-time through dependence analysis and cloning techniques in an object-based paradigm," M.S. Thesis, New Jersey Institute of Technology, May 1994.

[14] Reasoning Systems, Palo Alto, CA, "Refine Language Tools," 1993.

[15] C. Rich and R. Wills, "Recognizing a Program's Design: A Graph-Parsing Approach," *IEEE Software*, volume 7, number 1, 1990.

[16] H. Rubenstein, R. Piazza, and S. Roberts, "Separating Parsing and Analysis in Reverse Engineering Tools," *Proceedings of the Working Conference on Reverse Engineering*, May, 1993.

[17] C. H. Sampson, "Translating CMS-2 to Ada," *Proceedings of The Fourth Systems Reengineering Technology Workshop*, pages 143-156, Naval Surface Warfare Center, February 1994.

[18] A. L. Samuel, E. Sam, J. A. Haney, L. R. Welch, J. Lynch, T. Moffit, 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*, Naval Surface Warfare Center, February 1995.

[19] W. B. Scott, "Navy May Accelerate Missile Defense," *Aviation Week and Space Technology*, pp. 283-284, May 30, 1983.

[20] M. Sitaraman, L. R. Welch and D. E. Harms, "On Specification of Reusable Software Components," *The International Journal of Software Engineering and Knowledge Engineering*, volume 3, number 2, 1993.

[21] H. M. Sneed, "Economics of Software Re-engineering," *Software Maintenance: Research and Practice*, John Wiley and Sons, volume 3, number 3, Sept. 1991, pages 163-182.

[22] R. A. Steigerwald and L. R. Welch, "Reusable Component Retrieval for Real-Time Applications," *Proceedings of the First IEEE Workshop on Real-Time Applications*, May 1993.

[23] K. J. Stein, "Aegis Fleet Defense Nearing Sea Test," *Aviation Week and Space Technology*, pp. 32-35, August 13, 1973.

[24] K. J. Stein, "Aegis System Tested Successfully," *Aviation Week and Space Technology*, pp. 36-40, April 7, 1975.

[25] A. D. Stoyenko, L. R. Welch, and B. C. Cheng, "Response Time Prediction in Object-Based, Parallel Embedded Systems," to appear in *Euromicro Journal*, 1994, Special Issue on Parallel Processing in Embedded Real-Time Systems.

[26] "User Handbook for Macro Assembler," NAVSEA 0967-LP-598-8040, Sept. 1988.

[27] "User's Handbook for Navy Standard 16-Bit Computers Support Software," Vol. III, Hardware/Assembly/Fortran, NAVSEA 0967-LP-598-2030, April 1984.

[28] "User's Handbook for Navy Standard 16-Bit Computers Support Software," Vol. IV, CMS-2M, NAVSEA 0967-LP-598-2040, Feb. 1987.

[29] J. P. C. Verhoosel, L. R. Welch, D. K. Hammer, and A. D. Stoyenko, "Assignment and Pre-Runtime Scheduling of Object-Oriented, Hard Real-Time Parallel Processes Using Bead Partitioning," New Jersey Institute of Technology Technical Report CIS-93-16, December, 1993.

[30] J. P. C. Verhoosel, L. R. Welch, D. Hammer, and A. D. Stoyenko, "Assignment and Pre-Runtime Scheduling of Object-Based, Parallel Real-Time Processes," *IEEE Symposium on Parallel and Distributed Processing*, Oct. 1994.

[31] J. P. C. Verhoosel, L. R. Welch, D. K. Hammer, A. D. Stoyenko, and E. J. Luit, "A Formal Deterministic Scheduling Model for Object-Based, Hard Real-Time Executions," *Journal of Real-Time Systems*, **8(1)**, January 1995.

[32] L. R. Welch, "Assignment of ADT Modules to Processors," *Proceedings of the International Parallel Processing Symposium,* pages 72-75, March, 1992.

[33] L. R. Welch, A. D. Stoyenko, T. J. Marlowe, "Modeling Resource Contention among Distributed Periodic Processes," *Fourth IEEE Symposium on Parallel and Distributed Computing* (December 1992).

[34] L. R. Welch, A. D. Stoyenko, T. J. Marlowe, "Response Time Prediction for Distributed Periodic Processes Specified in CaRT-Spec," *Control Engineering Practice*, (in press).

[35] L. R. Welch, A. D. Stoyenko and S. Chen, "Assignment of ADT Modules with Random Neural Networks," *The Hawaii International Conference on System Sciences*, pages II-546-555, Jan. 1993.

[36] L. R. Welch, "Cloning ADT Modules to Increase Parallelism: Rationale and Techniques," *Fifth IEEE Symposium on Parallel and Distributed Computing*, pages 430-437, December 1993.

[37] L. R. Welch, "A Parallel Virtual Machine for Programs Composed of Abstract Data Types", *IEEE Transactions on Computers*, **43(11)**, Nov. 1994, pages 1249-1261.

[38] L. R. Welch, A. Samuel, M. Masters, R. Harrison, M. Wilson and J. Caruso, "Reengineering Complex Computer Systems for Enhanced Concurrency and Layering," *Journal of Systems and Software*, July 1995, (in press).

[39] L. R. Welch, G. Yu, J. Verhoosel, J. A. Haney, A. Samuel, and P. Ng, "Metrics for Evaluating Concurrency in Reengineered Complex Systems," *Annals of Software Engineering,* **1(1)**, Spring 1995.

[40] L. R. Welch, J. A. Haney, A. L. Samuel, R. D. Harrison, J. Lynch, M. W. Masters, T. Moffit, B. Ravindran, E. Sam, and W. Wright, "Reengineering of Legacy Systems: Toward an Automated Approach," *Proceedings of The Fifth Systems Reengineering Technology Workshop*, Naval Surface Warfare Center, February 1995.

[41] Xinotech Research Inc., Minneapolis, MN, "The Xinotech Program Composer 2.0," 1992.

[42] G. Yu and L. R. Welch. Program Dependence Analysis for Concurrency Exploitation in Programs Composed of Abstract Data Type Modules. In *Sixth IEEE Symposium on Parallel and Distributed Processing,* pages 66-73, October 1994.

[43] G. Yu and L. R. Welch, “A Novel Approach to Off-line Scheduling in Real-Time Systems,” *Informatica*, (in press).