New Jersey Institute of Technology
Digital Commons @ NJIT

Theses | Electronic Theses and Dissertations

5-31-1994

Extracting parallelism at compile-time through dependence analysis & cloning techniques in an object-based paradigm

Binoy Ravindran
New Jersey Institute of Technology

Follow this and additional works at: https://digitalcommons.njit.edu/theses

Part of the Computer Sciences Commons

Recommended Citation
Ravindran, Binoy, "Extracting parallelism at compile-time through dependence analysis & cloning techniques in an object-based paradigm" (1994). *Theses*. 1721.
https://digitalcommons.njit.edu/theses/1721

This Thesis is brought to you for free and open access by the Electronic Theses and Dissertations at Digital Commons @ NJIT. It has been accepted for inclusion in Theses by an authorized administrator of Digital Commons @ NJIT. For more information, please contact digitalcommons@njit.edu.

Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

NJIT
New Jersey's Science & Technology University

The Van Houten library has removed some of the personal information and all signatures from the approval page and biographical sketches of theses and dissertations in order to protect the identity of NJIT graduates and faculty.

ABSTRACT

EXTRACTING PARALLELISM AT COMPILE-TIME THROUGH DEPENDENCE ANALYSIS & CLONING TECHNIQUES IN AN OBJECT-BASED PARADIGM

by
Binoy Ravindran

The construct of Abstract Data Type (ADT) modules and Abstract Data Object (ADO) modules supported by most object-based languages are a great source for developing reusable code. To improve the run time performance of such object-based programs, we consider the asynchronous remote procedure call (ARPC) model of parallel execution, in which concurrency is achieved by having the caller and the callee (which are module instances) running on different processors. Frequently, an ADT module is needed simultaneously by other modules, thus causing contention. To resolve this, we clone the module instance in demand and distribute the copies across different processors, so that multiple clients can access the code concurrently. For identifying the facilities causing bottlenecks to the ARPC model, the dependence relations of the code is analyzed at compile-time. Instance dependences of the code are also analyzed in addition to conventional dependences to reveal the potential concurrency, and an upper bound on the number of clones of each facility that could be used in an application is determined. This parallelism information could be used by the assignment and the scheduling algorithms in the run time environment of the application for constructing a feasible real-time schedule, statically.

EXTRACTING PARALLELISM AT COMPILE-TIME THROUGH DEPENDENCE ANALYSIS & CLONING TECHNIQUES IN AN OBJECT-BASED PARADIGM

by
Binoy Ravindran

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Science

Department of Computer and Information Science

May 1994

APPROVAL PAGE

EXTRACTING PARALLELISM AT COMPILE-TIME THROUGH DEPENDENCE ANALYSIS & CLONING TECHNIQUES IN AN OBJECT-BASED PARADIGM

Binoy Ravindran

Dr. Lonnie R. Welch, Thesis Advisor — Date
Assistant Professor of Computer and Information Science, NJIT

Dr. James A. M. Mchugh, Committee Member — Date
Professor of Computer and Information Science
Associate Chairperson of the Department, NJIT

Dr. Andrew Sohn, Committee Member — Date
Assistant Professor of Computer and Information Science, NJIT

Blank Page

BIOGRAPHICAL SKETCH

Author: Binoy Ravindran

Degree: Master of Science in Computer Science

Date: May 1994

Undergraduate and Graduate Education:

- Master of Science in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 1994
- Bachelor of Technology in Mechanical Engineering,
  University of Kerala, India, 1991

Major: Computer Science

This Thesis is dedicated to
my niece Swetha who, with her birth
fostered a new generation in our family

ACKNOWLEDGMENT

I take this opportunity to express my deep gratitude to Professor Lonnie Welch, for his guidance, friendship and moral support throughout this work. It was an enlightening experience working under him.

My special thanks to Professors James Mchugh and Andrew Sohn for serving as members of the committee.

I am grateful to the U.S. Naval Surface Warfare Center for the funding of this project.

My appreciations are due to Gray, Scott, Jin, Manish and many other members of the group for their comments on the organization and contents of this manuscript. Many thanks goes to Pradeep and J. Roy for the timely help and suggestions they provided.

Finally, I express my sincere gratitude to my parents who. have always encouraged me to pursue higher avenues of learning.

TABLE OF CONTENTS

Chapter Page

1 INTRODUCTION . . . 1

1.1 Previous Work . . . 3

1.1.1 Program Dependence Analysis . . . 3

1.1.2 Cloning ADT Modules for Concurrency Enhancement . . . 4

1.2 Overview of the Thesis . . . 5

2 THE PROGRAMMING LANGUAGE & ASSOCIATED TOOLS . . . 6

2.1 The Language Model . . . 6

2.1.1 Program Definition . . . 6

2.1.2 Process Definition . . . 7

2.1.3 Module Class . . . 8

2.1.4 Mechanisms for Parameter Passing . . . 26

2.1.5 Compiling, Assembling & Linking . . . 29

2.2 Associated Tools . . . 30

2.3 An Application Program . . . 32

3 THE EXECUTION PARADIGM . . . 34

3.1 Introduction . . . 34

3.2 The Execution Model . . . 34

4 TECHNIQUES FOR CONCURRENCY EXTRACTION . . . 36

4.1 Asynchronous Remote Procedure Call . . . 36

4.1.1 The Program Call DAG . . . 38

4.1.2 Concurrency Propagation Techniques . . . 39

4.2 Cloning of ADT Instances . . . 43

4.2.1 Program Dependence Graphs . . . 43

4.2.2 Extracting Parallelism from Graphs . . . 57

Chapter Page

4.2.3 Cloning Analysis of the Application Program 58

4.2.4 Parallelism inside Loops 72

4.2.5 Interfacility Clone Analysis 78

5 ILLUSTRATION OF SYSTEM DESIGN 84

6 CONTRIBUTIONS TO KNOWLEDGE 86

APPENDIX A GRAMMAR OF RT-RESOLVE 87

APPENDIX B PRIMITIVE MODULE OPERATIONS 102

APPENDIX C VEHICLE APPLICATION PROGRAM 105

APPENDIX D CONDITIONAL TRANSFORMATIONS 119

REFERENCES 125

LIST OF TABLES

Table Page

4.1 Units of *coordsEqual* operation 71

4.2 Units Parallelism Matrix of *coordsEqual* operation 71

4.3 Groups of Units of *coordsEqual* operation 71

4.4 Facility Units Matrix of *coordsEqual* operation 72

4.5 Group Facility Matrix of *coordsEqual* operation 72

D.1 Units from Left Graph (Conditional at level 2) 120

D.2 Units Parallelism Matrix of Left Graph (level 2) 120

D.3 Groups of Units of Left Graph (level 2) 120

D.4 Facility Units Matrix of Left Graph (level 2) 120

D.5 Group Facility Matrix of Left Graph (level 2) 120

D.6 Units from Right Graph (Conditional at level 2) 120

D.7 Units Parallelism Matrix of Right Graph (level 2) 121

D.8 Groups of Units of Right Graph (level 2) 121

D.9 Facility Units Matrix of Right Graph (level 2) 121

D.10 Group Facility Matrix of Right Graph (level 2) 121

D.11 Units from Left Graph (Conditional at level 1) 122

D.12 Units Parallelism Matrix of Left Graph (level 1) 122

D.13 Groups of Units of Left Graph (level 1) 123

D.14 Facility Units Matrix of Left Graph (level 1) 123

D.15 Group Facility Matrix of Left Graph (level 1) 123

D.16 Units from Right Graph (Conditional at level 1) 124

D.17 Units Parallelism Matrix of Right Graph (level 1) 124

D.18 Groups of Units of Right Graph (level 1) 124

D.19 Facility Units Matrix of Right Graph (level 1) 124

Table	Page

D.20 Group Facility Matrix of Right Graph (level 1) 124

LIST OF FIGURES

Figure Page

2.1 An example *control process* 7

2.2 An example *process* 8

2.3 An example *module* 10

2.4 The *parameters* section of a *module* 12

2.5 The *auxiliary* section of a *module* 13

2.6 An example *facilities* section 14

2.7 An example *provided types* section (*Auxiliary section*) 15

2.8 An example *variables* section 16

2.9 An example *auxiliary operations* section 18

2.10 An example *auxiliary initialization* section 20

2.11 The *interface* section of a *module* 20

2.12 An example *provided types* section (*Interface section*) 21

2.13 An example *interface* section 27

4.1 Call DAG of the Vehicle Application Program 39

4.2 Extended Call DAG of the Vehicle Application Program 40

4.3 An if-statement (a) and its CDG (b) 44

4.4 The *coordsEqual* operation of *coordinate* module 46

4.5 The CDG of *coordsEqual* operation 47

4.6 The Statement Table of *coordsEqual* operation 48

4.7 Algorithm for building CDG 49

4.8 Algorithm for building DDG *(searching for data dependence)* 50

4.9 Algorithm for building DDG *(finding successive statement)* 51

4.10 Algorithm for building DDG *(finding common parameters)* 51

4.11 The DDG of *coordsEqual* operation 52

Figure | Page

4.12 Algorithm for building CDDG 53

4.13 The CDDG of *coordsEqual* operation 54

4.14 The PDG of *coordsEqual* operation 55

4.15 The FDG of *coordsEqual* operation 56

4.16 Algorithm for Finding Units (*main*) 59

4.17 Algorithm for Finding Units (*finding next statement*) 60

4.18 Algorithm for grouping Parallel Units 62

4.19 Algorithm for Transforming Conditionals (*main*) 63

4.20 Algorithm for Transforming Conditionals (*Depth-First-Search*) 64

4.21 Conditional Transformation of *coordsEqual* operation at level 2 65

4.22 Extracting Left Graph of Conditional (level 2) 66

4.23 Extracting Right Graph of Conditional (level 2) 67

4.24 Cloning Requirements of Conditional (at level 2) 67

4.25 Conditional Transformation of *coordsEqual* operation at level 1 68

4.26 Extracting Left Graph of Conditional (level 1) 69

4.27 Extracting Right Graph of Conditional (level 1) 69

4.28 Cloning Requirements of Conditional (at level 1) 70

4.29 Transformed DDG of *coordsEqual* operation 70

4.30 The procedure *AccessSeq* of module *pcomp* 73

4.31 DDG of operation *AccessSeq* (module *pcomp*) 74

4.32 DDG of the Loop (operation *AccessSeq*) 74

4.33 DDG of the Unrolled Loop 76

4.34 DDG of the Unrolled Loop with Facility & Antidependences 77

4.35 Algorithm for Handling Loops (*main*) 78

4.36 Algorithm for Handling Loops (*Unrolling Loop*) 79

4.37 Algorithm for Handling Loops (*Removing Antidependences*) 80

4.38 Algorithm for Handling Loops (*Adding Cross Iteration Fac. Dep's*) 81

Figure | Page

4.39 Algorithm for finding Direct-Clone-Requirements (DCR) 82
4.40 Algorithm for finding Transitive-Clone-Requirements (TCR) 83
4.41 Algorithm for finding Program-Clone-Needs 83
5.1 System Design of The Compiler at the *top level* or *level 1* 84
5.2 System Design of The Compiler at *level 2* 85
D.1 Left Graph of Conditional at level 2 119
D.2 Right Graph of Conditional at level 2 121
D.3 Left Graph of Conditional at level 1 122
D.4 Right Graph of Conditional at level 1 123

CHAPTER 1

INTRODUCTION

Software reusability has become a major issue primarily due to the crisis of increasing demand for new software systems and the inability of software engineers to keep pace with it. As a result of such a rapid demand, software engineers are eager to exploit the results of their previous development efforts leading to the reuse of code modules. One could easily argue that the vast majority of the code that exists today is not reusable. What gives much credence to this argument is that, ever since the software life cycle concept had been formulated, it has been found that most of the time and money is spent in software maintenance and most of that effort is spent in trying to determine what the code does. Reusing software components which have already proved their correctness or have already been debugged is obviously one way to reduce the development and maintenance cost. Improperly designed code. when attempted to reuse can create severe problems as it may have a form that makes them difficult to integrate into a system. Therefore many programmers and language designers recognize the need to develop modules with reuse in mind and thereby they frequently use the abstract data type (ADT) construct. An abstract data type component provides a collection of operations that can be invoked by other components. Use of ADTs lead to many benefits such as information hiding, encapsulation, loose coupling and high cohesion. All these are highly desirable properties for software reusability as they help to make software components easily adapt to different application environments. Most of the object-based languages support the constructs of abstract data types and abstract data objects (ADO). For example, Ada provides the generic package which are parametrized by types and operations. Also. C++ allows the definition of generic class templates which again when instantiated

with type and operation parameters gives rise to abstract data objects. However the cost of the reusability of programs constructed out of ADT and ADO modules is its low execution efficiency especially when these modules are highly generic and are parametrized by data types, thereby rendering the run time management highly expensive. Also the system performance deteriorates due to the cost of procedure calls, the communication overhead, and the encapsulation of the abstraction's data structures [10].

In this thesis, a parallel execution model (asynchronous remote procedure call, or ARPC) is considered to improve the performance of programs developed with ADTs and ADOs in a distributed and parallel system. In a distributed system, an abstract data type can be modeled as a *server* receiving requests for its operations from various *clients*. The server and its clients interact using the interprocess communication (IPC) primitives provided by the operating system and run on either the same or different machines. In such an environment, the server could be running on a dedicated processor and the clients would be invoking its operations via remote procedure calls. However, if multiple clients want to access their data variables managed by one server at the same time and only one client is granted access to the server, there will be contention for the server and all the other clients will have to wait until the server becomes available. To resolve this contention, the server code could be replicated and copies of the code (or *clones*) could be placed on different processors. By replicating the ADT facilities and distributing them across the various processing elements, multiple method calls could be served concurrently, thereby speeding up the execution of programs. Techniques have been developed [1] for identifying units of parallelism in programs composed of ADTs and for increasing parallelism by using replicated ADT instances. The programming paradigm used in this work consists of ADT and ADO templates, which form the basic reusable components.

To exploit parallelism from programs constructed out of ADT and ADO modules automatically, the dependence relations among method calls are analyzed. The classical data dependence graph (DDG) and control dependence graph (CDG) are extended to include facility (ADT instances) dependences or code dependences for the purpose of clone analysis. Algorithms presented in [1] have been implemented for determining the maximum number of clones of each ADT facility that can be used in an application.

In this section, we summarize the previous works on program dependence analysis and cloning techniques.

1.1 Previous Work

The work in this thesis is mainly on extracting parallelism information from programs constructed out of ADT and ADO modules and is based on two aspects:

- Program dependence analysis and
- Cloning of ADT modules.

In this section, previous research works on each of these areas is reviewed.

1.1.1 Program Dependence Analysis

The program dependence graph (PDG) is an intermediate representation of the data and control dependences between statements in a program. In the PDG, program statements are represented as nodes and directed edges denote the data and control dependences which the statements have with one another according to their lexical ordering in the source code. These dependence relationships determine the necessary sequencing between operations and can be used to expose potential parallelism in the program. Most of the previous works [2, 6, 8, 9] have used these dependences for code optimization and parallelism detection. However, data and control dependences

are not sufficient enough to represent relationships among statements in object-based programs, where the major activity is method calls. Call statements having neither data nor control dependences could be assigned to different processors and run in a parallel manner if no other dependence relations between the statements are revealed, as is the case here. However, there could be code dependence between statements if the statements call the same method, and this apparently could prevent such a concurrency. The code dependence relation therefore, can reveal the contention for the code of the shared method. None of the previous works has dealt with code dependence relations.

We introduce facility dependences into the program dependence analysis to reveal the contention between statements for common facilities. Two statements is said to have a facility dependence between them if they use methods provided by the same facility.

1.1.2 Cloning ADT Modules for Concurrency Enhancement

Previous research work on software component cloning has mainly been on compiler optimization and fault tolerance. Keith Cooper [2] uses cloning techniques for compiler optimization. His algorithm finds improvements in forward interprocedural data-flow solutions and clones those procedures that could lead to run time improvement.

In [6], replication (node splitting) is applied at the statement level to reduce communication and synchronization costs. Cloning ADT modules for exploiting parallelism has been addressed by Welch [1]. In his work, the contention for an ADT facility is revealed by partitioning the statements of an ADT module into *units*. A *unit* is defined as a sequence of one or more statements, which due to the data dependences among them, must execute in their lexical order. The statements of a unit cannot contend for a facility, but different units may. By further grouping the

units, an upper bound on the number of clones of facilities that could be used in an application is determined by a polynomial algorithm. Also, techniques have been presented to increase parallelism within loops by iteration unrolling, code motion, and removal of antidependences.

In this work, the PDGs have been extended to represent all kinds of dependences (data, control and facility dependences) and further, such an extended dependence graph is used to determine an upper bound on the number of clones of facilities that could be used.

1.2 Overview of the Thesis

The remainder of the thesis is organized as follows. In Chapter 2, an introduction to the language model and the programming paradigm assumed for the cloning analysis is described. The execution paradigm is described in Chapter 3 and is illustrated with an application program. Techniques for concurrency extraction forms the topic of Chapter 4. The ARPC model of parallel execution, theorems related to the facility dependence relations, and concurrency propagation techniques are discussed in this Chapter. The implementation (system design) of the dependence graph extractions and the cloning analysis is described in Chapter 5. Finally, we present the contributions of the work in Chapter 6.

CHAPTER 2

THE PROGRAMMING LANGUAGE & ASSOCIATED TOOLS

2.1 The Language Model

The construct of ADTs and ADOs are supported in most of the object-based languages like Ada, Modula-2, Clu and RESOLVE. The language model used in this work defines an application program to be composed of three distinct components: *program definition*, *process definition* and *class definition*. We explain each of these in the following sections.

2.1.1 Program Definition

The program definition is the main component in an application. It defines the processes that are to be instantiated and their timing constraints. The timing constraints of a process are the time parameters used by the run time system for invoking the process periodically in a real-time environment. The component is referred to as the *control process* of the application and has the following syntax:

control process:

    control process *process_name*

      begin

        <process_decl_sec>

      end *process_name*

<process_decl_sec>:

    <process_decl> | <process_decl_sec> <process_decl>

<process_decl>:

    process_name (deadline, frame);

Deadline and *frame* are the timing constraints on the process being defined. In a real-time environment, *frame* is the time period (interval) within which a process

```
control process main
  begin
    processA(100,200);
    processB(150,1000);
  end main
```

Figure 2.1 An example *control process*

activates, and *deadline* is its time deadline. An example of a control process is shown in Figure 2.1.

2.1.2 Process Definition

The definition of a process includes a *parameter section*, *variable declaration section*, *facility declaration section* and a *procedure declaration section*. The grammar is defined below.

<process>:

> process *process_name*
>
> { | process_parm }
>
> { | var_decl }
>
> { | fac_decl }
>
> process_proc_decl
>
> **end** *process_name*

The parameters of a process are the time constraints on it, as outlined in the previous section. Instantiation of a module creating *instances* or *facilities* is carried out in the *facilities* section. Variables local to the process if any, are declared in the variable declaration section. For any facility to be used in the procedure defined inside the process, it has to be instantiated first, in the facilities section. We explain the process of instantiation in detail in the next section. A process can have only a single procedure defined inside its procedure declaration section. Sequential

```
process stackdr
  facilities
    i is integer;
    s is stack(i.integer);
  end facilities
  procedure STACKDR
    begin
      local variables
        st1 : s.STACK;
        st2 : s.STACK;
        one : i.integer;
        five : i.integer;
        ten : i.integer;
      end local variables
        s.set_stack_size(st1, ten);
        s.set_stack_size(st2, ten);
        s.spush(st1,one);
        s.spush(st1,five);
  end STACKDR
end stackdr
```

Figure 2.2 An example *process*

execution of the application program actually begins with the first statement inside this procedure. The definitions of procedures and other subprograms supported by the language are outlined in subsequent sections. An example of a typical process is shown in Figure 2.2.

2.1.3 Module Class

The facilities discussed in the previous section, are instances of module templates which are ADT or ADO components. A typical ADT or ADO component in our language model exports a type that can be used to declare variables and has an interface section which provides a set of operations or methods. These operations can be used to manipulate (only) the variables which have been declared of the exported type. In other words, variables of the exported type of the ADT component can be accessed only through the provided methods. The ADT components or modules can

be defined to be generic i.e., they can be parametrized by types or by operations, and this generic nature of the component is what contributes to their reusability. To be used, modules must be instantiated. Instantiation of a module means fixing its parameters (actual) and choosing one of many implementations. Such an instance of a module is called a *facility.*

A module in the language model basically has three sections: the *parameter section*, the *auxiliary section* and the *interface section.* In addition to these, the number of operations or methods defined in the interface section is also explicitly stated at the beginning of the module. The module definition is shown below.

module:

> **module** *module_name*
>
> **num operations** = *int_literal ;*
>
> { | **<mod_parm_sec>**}
>
> { | **<aux_sec>**}
>
> { | **<intf_sec>**}
>
> **end** *module_name*

Example of a module is shown in Figure 2.3. The different sections of the module are detailed in the following subsections.

2.1.3.1 Parameters Section In the parameters section, parameters of the module are described, preceded and ended by the keywords *module parameters* and *end module parameters* respectively. The parameters of a module may include types and operations. A type parameter, is simply stated preceded by the keyword *type*. When a module is parametrized by an operation (a formal subprogram), the name of the subprogram, its parameters, parameter passing modes, return variable name and its type if any, are stated. Subprograms (operations or methods) in the language are either *procedures*, *functions* or *control functions.* We discuss the different methods

```
module EXAMPLE
  num operations = 2;

  module parameters
    ...
  end module parameters

  auxiliary
    ...
  end auxiliary

  interface
    procedure A
      ...
    end A

    procedure B
      ...
    end B
  end interface
end EXAMPLE
```

Figure 2.3 An example *module*

supported by the language in subsequent sections. The *parameters section* has the following definition:

<mod_parm_sec>:

module parameters

<mod_parm_seq>

end module parameters

<mod_parm_seq>:

<mod_parm> ; |

<mod_parm_seq> <mod_parm> ;

<mod_parm>:

type *type_name*

| <proc_hdr>

| <func_hdr>

| <ctrl_func_hdr>

The parameter section of a module parametrized by a type and an operation is shown in Figure 2.4. Also, notice that the parameter of the function *T_Copy* (i.e., *p*) is declared to be of type *T*, which in fact is a parameter type of the module itself. *Preserves* is a parameter passing mode; the different parameter passing mechanisms of the language is covered in a separate section. Note that the parameters section is optional, i.e., a module which is not parametrized, obviously need not require a parameters section.

2.1.3.2 Auxiliary Section

The definition of the *auxiliary section* is :

<aux_sec>:

auxiliary

{ | <fac_dec_sec> }

{ | <prvd_types> }

{ | <var_dec_sec> }

```
module parameters
  type T;
  function T_Copy returns x : T;
    parameters
      preserves p : T;
    end parameters
  end T_Copy;
module parameters
```

Figure 2.4 The *parameters* section of a *module*

```
    { | <aux_oper_dec_sec> }
    { | <real_aux_sec> }
  end auxiliary
```

We now discuss each of these sections separately. An example of an auxiliary section is shown in Figure 2.5.

2.1.3.3 Facilities Section Instantiation of modules creating *facilities*, is done inside the *facilities section*. This section is delimited with the keywords *facilities* and *end facilities*. The process of *instantiating* a module involves creating specialized copies of the module by fixing its formal parameters. The actual parameters being supplied to a module for instantiating it, could be even operations or types exported from other modules. Parameters exported by a module (operations, types) can be used only after instantiating the module (which exports them) and thereby creating a facility of it. In other words, to utilize any of the services provided by a module, an instance of it has to be created first. Once modules are instantiated (in the auxiliary section), the resulting facilities could be used in the operations defined inside the module. The syntax of the *facilities section* is shown below.

```
<fac_dec_sec>:
    <fac_dec> | <fac_dec_sec> <fac_dec>
<fac_dec>:
```

```
auxiliary
  facilities
    ...
  end facilities

  provided types
    ...
  end

  variables
    ...
  end variables

  operations
    ...
  end operations

  initialization
    ...
  end initialization
end auxiliary
```

Figure 2.5 The *auxiliary* section of a *module*

```
facilities
  i is integer;
  i1 is integer;
  a is array(T);
  r is record3(a.array, i.integer, i1.integer);
end facilities
```

Figure 2.6 An example *facilities* section

facility_name is *module_name* (<arg_list>) ;

<arg_list>:

arg_name , <arg_list>

2.1.3.4 Provided Types Section Types exported by a module if any, are stated in the provided types section. The keywords bounding the section are *provided types* and *end*. The definition is shown below:

<prvd_types>:

| <prvd_types_sec>

<prvd_types_sec>:

provided types

<prvd_types_seq>

end

<prvd_types_seq>:

<prvd_type> | <prvd_types_seq> <prvd_type>

<prvd_type>:

type_name is represented by *long_type_name ;*

Apart from stating the name of the exported type, its representation (which could be exported from another facility) is also stated, using the keywords *is represented by*. Note the distinction between types exported by a module and the type

```
provided types
  STACK is represented by r.record2;
end
```

Figure 2.7 An example *provided types* section (*Auxiliary section*)

with which it is parametrized. An exported type (from a module) can be used to declare variables (outside the module), and these variables can be manipulated only with the operations provided by the module. Direct Access to the data definition of the variable is not allowed and therefore any operation, if required to be performed on the variable has to be through the methods defined in the module exporting the variable's type. A parameter type on the other hand is a type imported by the module which is used to fix the formal type wherever it has been used inside the module. The provided types section of a module in which types exported by the module are stated, is shown in Figure 2.7. The illustrated auxiliary section also has a type which the particular module is exporting and note that the representation of this type is being exported from another instantiated module (a facility).

2.1.3.5 Variable Declaration Section This section contains the declaration of static facility variables of the module. These variables are quite similar to the global variables in other languages as it can be referenced in any operation declared inside the module. That is, variables declared in this section has a global effect within and inside the module (*only*). Initialization of the variables declared in this section takes place automatically when instances of the module (having this *section*) is created in other modules. The *variable declaration section* has the definition:

```
<var_dec_sec>:
        variables
          <var_dec_seq>
```

```
variables
  front: i.integer;
  rear: i.integer;
end variables
```

Figure 2.8 An example *variables* section

```
        end variables
<var_dec_seq>:
        <var_decl> | <var_dec_seq> <var_decl>
<var_decl>:
        var_name : type_name ;
```

2.1.3.6 Auxiliary Operation Declaration Section This section contains the declarations of operations, which have a local effect to the module. In other words, operations declared in the auxiliary section of a module can be called only by the operations declared in the interface section of the *same* module and not by any other module (operations) which declares a facility of it. The auxiliary methods therefore, are *à la* private methods. The syntax of the *auxiliary operation declaration section* is as follows.

```
<aux_oper_dec_sec>:
        | <aux_oper_dec_seq>
<aux_oper_dec_seq>:
        operations
          <oper_decl_seq>
        end operations
<oper_decl_seq>:
        <proc_decl> |
        <func_decl> |
```

<ctrl_decl> |

<oper_decl_seq>

<proc_decl> |

<oper_decl_seq>

<func_decl> |

<oper_decl_seq>

<ctrl_decl>

<proc_decl>:

procedure *proc_name*

< proc_parm_sec>

begin

<loc_var_dec_sec>

<code>

end *proc_name*

<func_decl>:

function *func_name* returns *var_name : type_name*

<func_ctrl_parm_sec>

begin

<loc_var_dec_sec>

code

end *func_name*

<ctrl_decl>:

control *func_name*

<func_ctrl_parm_sec>

begin

<loc_var_dec_sec>

code

end *func_name*

```
operations
  procedure set_stack_size
    parameters
      alters s: STACK;
      alters size: i.integer;
    end parameters
    begin
      local variables
        contents: a.array;
      end local variables
      r.rec1_access(s, contents);
      a.set_array_size(contents, size);
      r.rec1_access(s, contents);
    end set_stack_size

  control is_empty
    parameters
      preserves s : STACK;
    end parameters
    begin
      local variables
        top : i.integer;
        zero: i.integer;
      end local variables
      r.rec2_access(s, top);
      if i.equal(top,zero) then
        r.rec2_access(s,top);
        return true;
      else
        r.rec2_access(s, top);
        return false;
      end if;
    end is_empty
end operations
```

Figure 2.9 An example *auxiliary operations* section

The methods or operations (abstract data types) discussed above can be procedures, functions, or control functions. A procedure may modify its parameters whose modes are not *preserves*. We discuss the different parameter passing mechanisms of the language in a separate section. Functions and control functions may not modify their parameters. A function returns a value that must be assigned to a variable. A control function returns either *true* or *false*, which is used as the condition in an *if* statement or a *while* statement.

2.1.3.7 Real Auxiliary Initialization Section This section of the auxiliary section of a module contains the code which has to be executed first when a facility of the module is created. The syntax of the *real auxiliary initialization section* is as follows.

```
<real_aux_sec>:
        | <real_aux_init>
<real_aux_init>:
        initialization
          begin
            { | <var_dec_sec> }
            <code>
          end initialization
```

By default every module contains an implicit initialize operation which contains code to initialize the facilities and static variables declared in the module. This code for initialization is inserted by the compiler. However if the user desires any variables to be initialized, then that could be stated explicitly in the real auxiliary initialization section. The compiler would include the user specified initialization operations with the default ones.

```
initialization
  begin
    local variables
      front : i.integer;
    end local variables
    i.increment(front);
end initialization
```

Figure 2.10 An example *auxiliary initialization* section

```
interface
  type type_name ...........
  end type_name

  procedure A
    ...
  end A

  function B
    ...
  end B
end interface
```

Figure 2.11 The *interface* section of a *module*

2.1.3.8 Interface Section The methods in a module are defined in the interface section. The interface section has a *type declaration* section and an *operation declaration* section. The section has the form:

```
<intf_sec>:

        interface
          { | <type_decl_seq> }
          { | <opr_decl_seq> }
        end interface
```

The interface section of a module is shown in Figure 2.11. We now explain each of these sections separately.

```
type STACK is represented by r.record2 exemplar ex
end STACK
```

Figure 2.12 An example *provided types* section (*Interface section*)

2.1.3.9 Provided Types The provided types are declared in the *type declaration* section and has the syntax:

<type_decl_seq>:

<type_decl> | <type_decl_seq> <type_decl>

<type_decl>:

type *type_name* is represented by *long_type_name*

exemplar *var_name*

{ | <type_init> }

{ | <type_fin> }

end *type_name*

<type_init>:

initialization

begin

{ | <var_dec_sec> }

<code>

end initialization

<type_fin>:

finalization

begin

{ | <var_dec_sec> }

<code>

end finalization

As indicated in the *type declaration* section, each type in a *synthesized* module is represented by another type which is a parameter to the module or a type provided by some other facility instantiated in the module. A *synthesized* module therefore, is a module built with types exported from other modules (while being instantiated), as opposed to *primitive* ones which are totally *built-in* i.e., provided at the language level. The *type_init* section contains the code to be executed when a variable of the declared type is initialized. The *exemplar* is initialized at the beginning of the execution of the operation by calling the initialization operation of the representation type. Local variables may be declared and the statements (code) may modify the initial value given to the exemplar. The *type_init* section is optional and if it is not specified by the user, the compiler still would generate code for the operation, which contains calls to the initialization operation of the representative type.

The *type_fin* section contains the code to be executed when a variable of the declared type is finalized. The exemplar is finalized at the beginning of the execution of the operation by calling the finalizing operation of the representative type. Like the *type_init* section, this section is also optional and if not specified by the user, the compiler as before, generates code which incorporates calls to the finalization operation of the representative type.

2.1.3.10 Interface Operation Declaration Section This section is quite similar to the *auxiliary operation declaration section*. However, unlike in the *auxiliary section*, the operations defined in the *interface operation declaration section* can be called by any external module which has an instance of the module with called operation in it (*à la* public methods). Also, note that the interface operations are the operations which a module exports to other modules. The syntax of the operation definition in this section is the same as that of its counterpart section

in the auxiliary. We now discuss the executable statements or instructions of the language.

2.1.3.11 Code The definition of a method (as discussed) includes declaring its parameters and local variables, followed by the actual code wherein the major activity is instance calls, as in most other object-based languages. The code (executable statements) of the language has a syntax:

<code>:

  <stmt> | **<code> <stmt>**

<stmt>:

  <swap> |

  <assign> |

  <if> |

  <while> |

  <return> |

  <do> |

  <proc_call>

As indicated, the different type of statements supported in the language are *swap*, *assign*, *if*, *while*, *return*, *do* and *procedure calls*. Except for the *swap* statement, the other operations are common features in all programming languages. We now discuss each of these statements separately, in the following sections.

2.1.3.12 Swap Statement The only built-in primitive for manipulating the values of variables is the *swap* statement, which simply exchanges the values of the two variables (i.e., the operands involved). The swap operator is denoted by :=:. For example, to swap the values of a and b, one would write a :=: b. The statement has the form:

<swap>:

var_name :=: *var_name*

2.1.3.13 Assignment Statement The statement has the syntax:

<assign>:

var_name := *func_call*

The assignment statement in our language model, unlike in other languages does not support copying of one variable to another. Thus, one cannot write *a:=b*. To achieve a copy, one must explicitly call the copy function: *a:=integer_copy(b)* . In fact, assignment statement in the language, assigns the return value of a function call to a variable. For copying the value of one variable to another, a call to the copy function of the module providing the variable's type must be made.

2.1.3.14 If Statement The statement has the syntax:

<if>:

```
if { | not } <ctrl_call> then
   <code>
{ | else
   <code> }
end if
```

If statements *always* contain a *control call* which returns a boolean value. The problem of "dangling else" cannot occur because of the explicit *end if*.

2.1.3.15 While Statement The statement has the form:

<while>:

```
while { | not } <ctrl_call> do
```

```
    <code>
  end while
```

Like the *if* statements, the *while* statements of the language also, always contains a *control call*.

2.1.3.16 Do Statement The statement has the syntax:

<do>:

```
do count times
  begin
    <code>
  end do
```

count is an integer constant and as implied, the loop is executed *count* number of times.

2.1.3.17 Return Statement The statement has the syntax:

<return>:

```
return |
return true |
return false
```

return true and *return false* can be used only in *control functions* to return a boolean value. However *return* can be used in any operation for an unconditional return from it.

2.1.3.18 Procedure Call The statement has the following definition:

<proc_call>:

```
long_proc_name { | ( arg_list ) }
```

long_proc_name is similar to *long_name* i.e., it signifies that the called procedure can be

- Provided by the module itself,
- Provided by an instantiated facility or
- Parameter to a module.

The *arg_list* specifies the parameters to the procedure. *func_call* and *ctrl_call* are quite similar to the *proc_call* except for the difference in the parameter passing modes as discussed previously. An example of the interface section is given in Figure 2.13.

2.1.4 Mechanisms for Parameter Passing

Conceptually, parameters are passed by *swapping* i.e., at operation invocation, the values of the formal parameters are swapped with the values of actual parameters; and on operation return, they are swapped again. Any implementation of parameter passing that achieves this abstract effect is, of course, acceptable. As discussed in [4], component efficiency increases when the values of composite data structures are swapped instead of copying them. The arguments to a call must be unique, i.e., the same variable may not appear twice in a particular argument list.

The different parameter passing modes are defined below:

1. *Alters:* The value of the actual parameter is modified. Information flows from the caller to the callee at invocation and flows in the reverse direction upon return.

2. *Preserves:* The value of the actual parameter may be modified, but is restored to its original value before the operation returns. Information flows from the caller to the callee at invocation and the *same* information flows in the reverse direction upon return.

```
interface
  type QUEUE is represented by r.record3 exemplar ex
  end QUEUE

  procedure setsize
    parameters
      alters Q; QUEUE;
      alters size: i.integer;
    end parameters
    begin
      local variables
        contents: a.array;
      end local variables
      r.rec1_access(Q, contents);
      a.set_array_size(contents, size);
      r.rec1_access(Q, contents);
    end setsize

  control IsEmpty
    parameters
      preserves Q: QUEUE;
    end parameters

    begin
      local variables
        front: i.integer;
        rear: i.integer;
      end local variables
      r.rec3_access(Q, rear);
      if i.equal(rear,front) then
        return true;
      else
        return false;
      end if;
    end IsEmpty
end interface
```

Figure 2.13 An example *interface* section

3. *Consumes:* The value of the parameter passed to the operation is "consumed" by the procedure. Information flows only from the caller to the callee. An initial value is assigned to the actual parameter upon return.

4. *Produces:* Is used to provide the caller with a value created by the operation. Information flows only from the operation to the caller. The actual parameter is finalized before the new value is assigned.

Local variables are automatically initialized (by allocating storage and giving a value to the contents of the storage) upon entry to an operation that declares them, and finalized (by reclaiming storage) upon exit from an operation that declares it. A call to initialize (or finalize) a variable is inserted by the compiler at the beginning (or end) of the code of the operation that declares it. The language provides the types *integer* and *array of integer*. Variables are automatically assigned initial values. Integer variables are assigned the initial value of zero (0). Integer arrays are initialized to have sizes of zero.

Additional features of the language include the complete absence of global variables. Instead, operations can access three kinds of data: operation parameters, local variables and module variables (static variables associated with a module instance that are shared among operations exported by that instance). Aliases cannot occur, i.e., the data structure representing a variable's value can only be known by one name at any time. No types are built into the language, therefore almost all statements are procedure calls, since manipulating a variable's value can take place only by a call to the facility operation exporting the variable's type. Modules cannot be instantiated dynamically, i.e., instantiations of modules are declarations (the analogy could be that of the variable-type relation i.e., an *instance* is to a *module* what a *variable* is to its *type*) that occur outside the code of the module operations and all instantiations are performed when a program begins execution.

The operations for manipulating integer and array variables are automatically defined and should not be redefined in a user's program. The complete grammar of the language is given in Appendix A and the different integer and array operations provided at the language level are defined in Appendix B.

2.1.5 Compiling, Assembling & Linking

An application is developed in the proposed language model through separately written and compiled modules. Separate compilation of modules is a feature of our language model and this enables to develop programs in a highly modular fashion, contributing much to an *off the shelf* style of programming. The compiled modules are then assembled (also done separately), before being linked together by the linker and loaded.

The compiler expects the module files to be named with the name of the module itself. That is, a file containing a module say, *queue* has to be named *queue* itself and this naming convention has been standardized with the language associated tools also, which we discuss in the next section. Also, note that the source code of a module has to be contained in a single file. The compiler doesn't support the spreading of a module code across multiple files. The compiler is invoked by the name *CR*, and to compile a source module:

$CR *module_file*

The compiler generates a set of four files:

- *module_file.asm*
- *module_file.fac*
- *module_file.gpd*
- *module_file.xtrn*

These are needed by the assembler and the linker. Once compiled, the same source module files are then assembled as:

$assem *module_file*

The assembler generates the machine code in a file, *module_file.mac*. The linker is then invoked to link all the assembled files. The argument to the linker is just the file name of the *control process* module say, *ctl_file* which is the root module in the application. Note that this file *ctl_file* also must be compiled and assembled as any other module in the application. The linker is invoked as:

$linker *ctl_file*

The linker produces the files:

- *ctl_file.code*
- *ctl_file.proc*
- *ctl_file.exe*
- *ctl_file.disp*

Once these files are produced, the application is ready to be loaded onto the run-time system. The run-time system is then invoked as:

$rtss *ctl_file*

2.2 Associated Tools

The language associated tools developed as part of this work and otherwise, includes a *DAG Generator* and a *Graph & Clones Extractor*. The DAG Generator generates the *Call DAG* and the Graph & Clones Extractor extracts the *program dependence graphs* of an application. We explain the *Call DAG* and the *dependence graphs* in Chapter 4. The DAG Generator takes the file name of the *control process* module as the argument and generates the Call DAG of the application. It is invoked as:

$daggen *ctl_file*

The *daggen* produces the following files describing the Call DAG.

- *dict.dag*
- *edges.dag*
- *obj.dag*
- *proc.dag*

The Graph & Clones Extractor generates the dependence graphs and the cloning needs of facilities for each operation of a module in the application. It is therefore invoked with module file as the argument, as:

$graphgen *module_file*

The graphgen generates the files:

- *module_file.cdg*
- *module_file.ddg*
- *module_file.cddg*
- *module_file.fdg*
- *module_file.pdg*
- *module_file.clone*

The "*._dg*" files describes the different dependence graphs and the *module_file.clone* details the facility-clone needs of the module.

2.3 An Application Program

In this section, a real-time application called *vehicle* developed in our language model is selected and explained, as an example to illustrate the programming paradigm.

The application program basically uses six modules, each declared before its use. These are the *main module*, *process vehicledr*, *integer*, *vehicle*, *coordinate* and *record2*. All the modules (except for the primitive ones, *integer* and *record2*) are illustrated in Appendix C. The main module is the *control process main* and it calls a single process, the *vehicledr*.

The process *vehicledr* uses the facilities *i*, an instance of the module *integer* and *v*, which is an instance of the module *vehicle*. The process also incorporates the procedure definition *vehicledr* in it, which has its own set of local variables and most of them have been declared to be of the types exported from other modules. The facility *i*, an instance of the *integer* module exports the type *integer* and *v*, an instance of the *vehicle* module exports a user - defined type: *vehicletype*. As illustrated, the code in the *vehicledr* procedure is mostly call statements, invoking operations defined in facilities *i* and *v*. The *integer* module is a primitive module provided at the language level for integer operations and is used for manipulating integer variables.

The *vehicle* module is defined and compiled separately. It illustrates a typical module of the language which is parametrized by a type, *vehicleType*. Note that the *vehicleType* defined in the auxiliary section is itself an instance of a type exported from another facility, *re*. The facility *re* is an instance of the module *record2* and is instantiated with the parameters *in.integer* and *co.coordtype* which again are exported from the respective modules. The operations defined in the interface section of the *vehicle* module further illustrates the parameter passing modes, the facility instantiations, the mechanism of type exporting etc., in our object-based paradigm. We use

the vehicle application program throughout this thesis for illustrating the execution paradigm, ARPC model and the concurrency propagation techniques.

CHAPTER 3

THE EXECUTION PARADIGM

3.1 Introduction

The execution paradigm is explained in this Chapter, based on the vehicle application program which was illustrated in Chapter 2. The main module, *control process main* acts as an informer to the compiler and the linker, informing the system about the process *vehicledr* that has to be instantiated with the actual parameters, which are its timing constraints. An instance of the process *vehicledr* is then created by the linker and the execution of the program begins with a call to the operation *gen_one*, which is the first executable statement in it. Within the process *vehicledr*, the instantiation of the modules *integer* and *vehicle* takes place to create the facilities *it* and *v* respectively. Facility variables are declared in the procedure *vehciledr* using the types provided by *i* and *v*. Operations of the facilities *i* and *v* are called in the procedure *vehicledr* of the process *vehciledr*, using the notation: *facility.operation(parameters)*.

In the *vehicle* module, instantiation of *integer*, *coordinate* and *record2* modules takes place, creating the facilities *in*, *co* and *re* respectively. The type *integer* provided by the facility *in* and *coordtype* provided by the facility *coordinate* are used to instantiate the *record2* module, creating the instance *re*. Further, the type *record2* provided by the facility *re* is exported as the type *vehicleType* of the module *vehicle* itself.

3.2 The Execution Model

Sequential execution of the application program proceeds as follows. The initialization operation of a facility invokes the facility initialization operations of all facilities it instantiates; initializes its facility variables; and executes the user-defined facility initialization code. Execution begins when the facility initialization

operation of the main facility i.e., process *vehicledr* is invoked by the run-time system. We denote the facility initialization operation of this process as *vehicledr.minit* and a similar convention is adopted with all the other facilities. The operation *vehicledr.minit* invokes *i.minit* and *v.minit*. Since *v* creates facilities *in*, *co* and *re*, its facility initialization operation, *v.minit* invokes *in.minit*, *co.minit* and *re.minit*. Further, *co* invokes *i.minit*, *in.minit* and *re.minit*. Notice that we have different instantiations of the modules *integer*, *record2* in different as well as same modules. Thus the *minit* operation of each facility initializes the module instances created by that facility.

After *vehicledr.minit* initializes the facilities declared in *vehicledr*, it initializes the facility variables of *vehicledr* (*veh1*, *veh2*, *id1* etc.) by calling the type initialization operations of facilities *i* and *v*. We denote the type initialization operation for *typei* provided by facility *p* as *p.typeitinit*. Similarly, *p.typeitfin* will denote the type finalization operation for *typei* provided by facility *p*. The initialization of variables *veh1*, *veh2*, *id1* etc., is therefore accomplished by invoking *v.type1tinit*, *v.type1tinit*, *i.type1init* respectively and so on.

The language is implemented by having each type initialization operation return a pointer to the *representation* of a variable, storing the pointer in the activation record of the operation that declared the variable. When the variable is passed as a parameter, only the pointer is passed. Since information hiding is enforced by the language, such a pointer will only be dereferenced by an operation of the facility providing the variable's type; operations of facilities other than the one providing the variable's type can only pass the pointer to other operations.

Once the facility variables of *vehicledr* have been initialized, the user-defined code of the facility initialization operation is executed. Thus, procedure *vehicledr* being the facility initialization operation of *vehicledr*, *i.gen_one* is called, then *i.increment* is called and so on.

CHAPTER 4

TECHNIQUES FOR CONCURRENCY EXTRACTION

4.1 Asynchronous Remote Procedure Call

In this section, the parallel execution model proposed for the execution of programs constructed out of ADTs is discussed. Architecture for Reusable software Components (ARC) [5] is an environment which has been developed for the execution of ADT modules supporting reusability, taking into account the potential run time inefficiencies of such software. In the distributed memory, parallel computing environment assumed for our execution paradigm, ARC is used as the basic processing element.

In the proposed model, programs are executed in parallel as follows. (Refer to the Vehicle Application discussed in Chapter 2). The code of the facilities is statically assigned to the PEs (Processing Elements) and multiple facilities may reside on the same PE. Execution of the program begins when the facility initialization operation of the main facility *vehicledr* is invoked by the run time system. The operation *vehicledr.minit*, then invokes the initialization operations of all other facilities instantiated in it; *i.minit* and *v.minit*. The execution of these operations which had been called by *vehicledr.minit* proceeds in a parallel fashion, if the facilities *i* and *v* are residing on different PEs. Also, since the facility *v* creates or instantiates the facilities *in*, *co* and *re*, its initialization operation *v.minit* invokes *in.minit co.minit* and *re.minit*. Similarly, the *minit* operations of each facility initializes the module instances created by that facility.

After *vehicledr.minit* initializes the facilities declared in *vehicledr*, it then initializes the variables *veh1*, *veh2*, *id1* etc., declared in *vehicledr* by calling *v.type1tinit*, *v.type1tinit*, *i.type1init* respectively and so on. Initialization of a variable involves storage allocation and assigning an initial value to the allocated storage. Thus, a variable's representation is stored on the PE where the code of

the type initialization operation that creates it resides. Therefore the only way the pointer to the variable can be used (by operations of facilities other than the one which provides the variable's type) is by passing it as a parameter. Since information hiding is enforced by the language, such a pointer will only be dereferenced by an operation of the facility providing the variable's type and such operations will reside on the same PE as the representation of the variable; operations of facilities other than the one providing the variable's type can only pass the pointer as a parameter to other operations.

Once the facility variables of *vehicledr* have been initialized, the user-defined code of the facility initialization operation is executed. Thus operations *gen_one*, *increment* etc., are called in their lexical order.

When a variable is passed as an argument in a call, the implementation ensures that only a pointer to its representation is passed. Thus there exists little communication overhead for calls. Also, to maintain consistency, only a single copy of the pointer to a data structure is accessible at any instant. 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 parallel execution is termed Asynchronous Remote Procedure Call or, ARPC. A variable is automatically locked when it is passed as a parameter to a call and is unlocked upon return of the call. Any operation attempting to access a locked variable must wait for a remote call to return (and then unlock the variable) before retrying to access.

The ARPC model can achieve parallel execution at multiple levels in the abstraction hierarchy. Thus potential parallelism within a program increases with the number of levels of abstraction and the model encourages the development of highly cohesive, loosely coupled modules.

4.1.1 The Program Call DAG

In this section, the construction of the Directed Acyclic Graph (DAG) is illustrated, which can be used to model the potential parallelism in a program. The DAG for a particular program shows the relationship among its distributable components, and the maximum amount of parallelism attainable with ARPC. The graph can be used for assigning the facilities on to the PEs.

A program is modeled by the DAG, G = (V,E), where:

- v ∈ V denotes the operations of a facility, f(v);
- (x,y) ∈ E indicates that the code of facility f(x) calls some operation(s) provided by facility f(y); and
- There exists exactly one vertex in G with indegree 0, representing the facility at the highest level of the abstraction hierarchy. This vertex is referred to as G.root.

The DAG representing a particular program can be constructed as follows.

1. Place a vertex in the graph for each facility used in the program.
2. Place an edge in the graph for each call dependency in the program. Only calls between operations of different facilities are represented in the graph.

The DAG for the sample program (shown in Figure 4.1) contains a node for each module instance used. The node *control process main* in the graph indicates the root module invoking other modules in the program. Edges between nodes denote calls to operations of one facility by another facility. As an example, *vehicledr* process calls operations of facilities *i* and *v* and so on. Also, note the flow of edges in the DAG between siblings, indicating call relationship between facilities at the same level. This is due to the instantiation of one facility using the types and/or operations provided by other facilities at the same level. For example, in the *vehicle* module, an instance

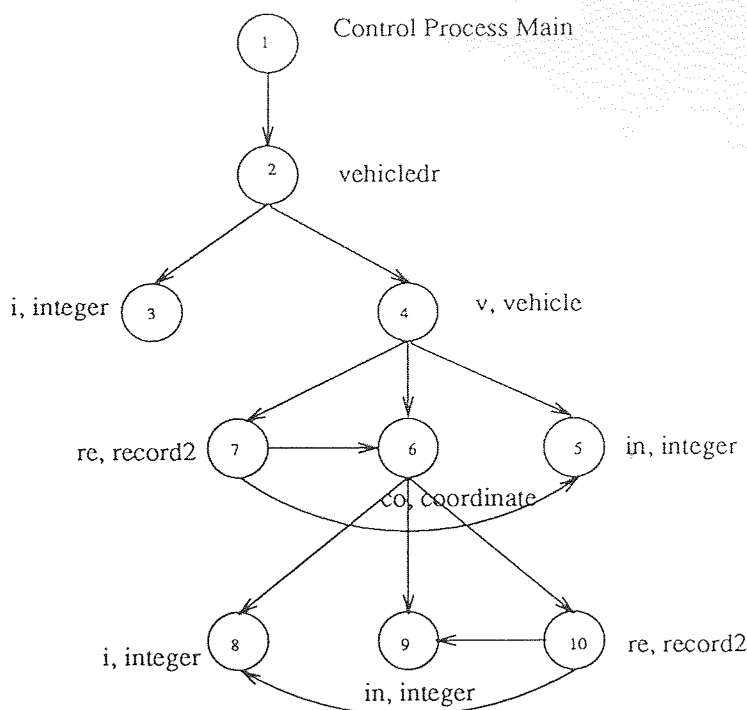


Figure 4.1 Call DAG of the Vehicle Application Program

of the *record2* module (*re*) is created using types exported from instances of *integer* (*in*) and *coordinate* (*co*) modules.

4.1.2 Concurrency Propagation Techniques

In this section, the ARPC model is evaluated and theorems identified in the context of concurrency propagation and parallelism extraction are discussed. Before the proposed theorems are formally stated, the terminology used is first elaborated. The term *chain* is defined as a sequence of facility names: $a \circ b \circ \cdots \circ n$, where f immediately preceding g in the sequence indicates that an operation of f calls an operation of g. A chain basically denotes a calling sequence that occurs in the source code of a program. For example, the chain $a \circ b \circ e$ signifies that an operation of facility a calls an operation of facility b, that an operation of facility b calls an operation of facility e. The chain also indicates the execution of an operation of the last facility named in the chain. Thus, the chain $a \circ b \circ e$ represents the state in which an operation of facility e is executing as a result of a call from an operation

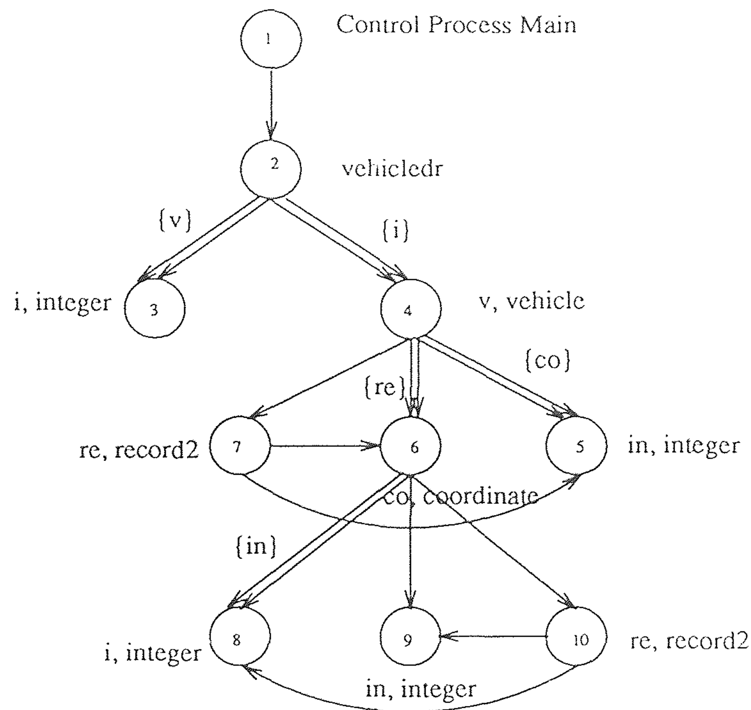


Figure 4.2 Extended Call DAG of the Vehicle Application Program

of facility b (which in turn was called by an operation of facility a). Operations of facilities a and b may or may not be executing in parallel with the operation of c, depending upon synchronization constraints; the chain does not specify these facts.

In the remainder of this Chapter, greek letters $(\alpha, \beta, \gamma, \ldots)$ are used to specify chains, and lower case English letters (a, b, c, ...) are used to denote facility names and operation names.

4.1.2.1 The Extended Call DAG

The program Call DAG described previously could be extended to demonstrate two kinds of parallelism relationships.

In the extended Call DAG, all pairs of facilities say (a,b) where $a.p$ calls $b.q$ and where $a.p$ can continue its execution after calling $b.q$ because there are no common parameters between the two (call) statements, are represented in the Call DAG as an edge drawn using parallel lines. As as an example, if $(a,b) \in E$, then a can execute in parallel with b if $\exists p$, q such that

1. $a.p$ calls $b.q$ and

2. the call is immediately followed by at least one statement that does not access any of the parameters passed to q.

The extended Call DAG for the example application program is shown in Figure 4.2. The parallel edge between the nodes *vehicledr* and *integer* in the graph indicates that the process *vehicledr* can continue with its execution even after calling i, an instance of the *integer* module, at least in one case, because of the absence of common parameters.

The Call DAG can also be used to indicate which immediate descendants of a vertex can execute in parallel with each other by placing labels on the edges. For example, suppose that $(a,b) \in E$, representing a call from operation $a.p$ to operation $b.q$; and $(a,c) \in E$, representing a call from operation $a.p$ to operation $c.r$. Assume the call to q is immediately followed by a call to r, and that the two calls have no parameters in common. Using the ARPC model, the execution of q can proceed in parallel with the execution of r. Such parallelism between facilities is denoted as labels on the edges (a,b) and (a,c). The labels are sets of facilities. Thus the labels on the edge (a,b) is c and the label on edge (a,c) is b. In the Extended Call DAG shown in Figure 4.2, the label v on the edge $(vehicledr,i)$ indicates that the process *vehicledr* can continue its execution by calling the facility v, even after invoking a call to the facility i.

4.1.2.2 Theorems for Concurrency Propagation

The fact that all chains begin with the same facility is true since a single sequential program is being parallelized, and only a single chain executes initially. Thus for any two chains α, β of an application program, it is true that they have a common prefix. This is formally stated in Theorem 1.

Theorem 1 For any two chains α, β of a program, it is true that $\exists \gamma, \delta, \epsilon(\alpha = \gamma \circ \delta \wedge \beta = \gamma \circ \epsilon)$

Assume that an operation p of facility a (denoted as $a.p$) calls an operation q of facility b (denoted as $b.q$), and that q calls an operation r of facility c (denoted as $c.r$). Let α represent an arbitrary chain. If the following are true,

1. the chain $\alpha \circ$ a can execute in parallel with the chain $\alpha \circ$ a $\circ$ b.

2. the chain $\alpha \circ$ a $\circ$ b can execute in parallel with the chain $\alpha \circ$ a $\circ$ b $\circ$ c.

then it is also true that the chain $\alpha \circ$ a can execute in parallel with the chain $\alpha \circ$ a $\circ$ b $\circ$ c. Intuitively, this means that if $a.p$ can execute in parallel with its call to $b.q$, and if $b.q$ can execute in parallel with its call to $c.r$, then $a.p$ can execute in parallel with the call of $b.q$ to $c.r$. This fact is formally stated as Theorem 2. The symbol $\|$ when placed between two chains denotes that the chains can run in parallel.

Theorem 2 if $\alpha \parallel \alpha \circ$ a $\wedge$ $\alpha \circ$ a $\parallel \alpha \circ$ a $\circ$ b then $\alpha \parallel \alpha \circ$ a $\circ$ b

Theorem 2 is used in the assignment algorithms (assigning modules to processors) discussed in [3]. The $\|$ relation is not transitive. That means, for some arbitrary chains say, α, β and γ, if $\alpha \parallel \beta \wedge \beta \parallel \gamma$ is true, then $\alpha \parallel \gamma$ need not be true. To further illustrate this, consider the case where $a.p$ calls $b.q$ and $a.p$ calls $c.r$. It will be true that $\alpha \circ$ a $\parallel \alpha \circ$ a $\circ$ b if after the call to q, $a.p$ executes code (which may be a call statement like $c.r$) which does not access parameters passed to q. For example, $a.p$ may call r with different parameters than that were used in the call to q. However following the call to r, p may access one of the parameters passed to r. Such an access can cause p to wait until r returns. Thus $\alpha \circ$ a $\parallel \alpha \circ$ a $\circ$ b $\wedge$ $\alpha \circ$ a $\circ$ b $\parallel \alpha \circ$ a $\circ$ c, but $\neg(\alpha \circ$ a $\parallel \alpha \circ$ a $\circ$ c).

Theorem 3 deals with the parallel execution of chains. It states that, if two chains α and β can execute in parallel, then chains $\alpha \circ$ a and $\beta \circ b$ can also execute in parallel as long as a does not represent the same facility as b, and a is not used in

chain β and b is not used in chain α. For example, if a $\circ$ b $\|$ a $\circ$ c then a $\circ$ b $\circ$ d $\|$ a $\circ$ c $\circ$ e ,but $\neg$(a $\circ$ b $\circ$ d $\|$ a $\circ$ c $\circ$ d), and $\neg$(a $\circ$ b $\circ$ d $\|$ a $\circ$ c $\circ$ b). The theorem is formally stated as follows.

Theorem 3 if $\alpha \| \beta$ then $\alpha \circ$ a $\| \beta \circ$ b, if all of the following are true:

1. a $\neq$ b
2. a is not in the chain β
3. b is not in the chain α

4.2 Cloning of ADT Instances

The amount of potential parallelism inherent in the program is fully revealed by analyzing the dependence relations of the source code. As discussed in Chapter 1, we extend the dependence relations of the program to include facility dependences (or instance dependences), since that could identify greater opportunities for exploiting parallelism. In this section, the identification of such opportunities through dependence analysis and the constraints to the ARPC model are discussed. We begin with the program dependence graphs.

4.2.1 Program Dependence Graphs

The relation among statements in the program is represented by the program dependence graphs. In the program dependence graphs, statements are represented as nodes and edges denote the dependences between them as implied by their lexical order. The basic dependences among the statements are control and data, and this results in the Control Dependence Graph (CDG) and the Data Dependence Graph (DDG). The dependences among program statements due to facilities are represented in the Facility Dependence Graph (FDG) and in the FDG, an edge indicates that the source and the destination use the same facility. Each of these

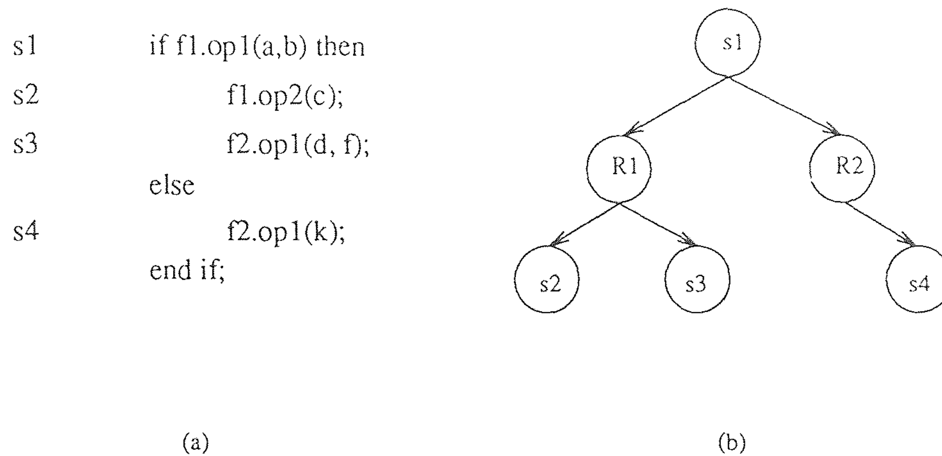


Figure 4.3 An if-statement (a) and its CDG (b)

dependences are defined in the following sections and also, it is shown how these graphs are generated at compile-time for the cloning analysis to follow.

4.2.1.1 Control Dependence *For any two statements S_i and S_j, if S_j has to be executed after S_i because of the control structures of the language (such as if-statements, while-statements), then the statement S_j is said to be* **control dependent** *upon statement S_i.*

For example, in an if-statement structure, all the statements in the two branches of the conditional must wait for the completion of the if-statement which is the evaluation statement, before the execution could continue any further. Therefore all the statements in the two branches of the if-statement are control dependent upon the conditional evaluation statement.

A **control dependence graph (CDG)** *is a directed acyclic graph (DAG) in which nodes represent program statements and edges, control dependences between them.* Formally,

$$CDG(S_i, S_j) = \begin{cases} 1, & \text{if } S_i \rightarrow^c S_j; \\ 0, & \text{otherwise}; \end{cases}$$

The CDG could be built in different ways as outlined in [2, 6]. We build the CDG from another graph called the Statement Table which contains all pertinent information about each statement in the program. The attributes of program statements like *statement type, statement dependence nesting level, statement address, facility used, parameters* etc., are stored in the statement table and such a graph is easily generated from the compiler. A statement called *entry* is added to the CDG for convenience and it just means that all statements in the CDG are directly or indirectly control dependent upon *entry*, and no statements could be executed without executing this *entry* node. Also, for a statement which has two or more branches, 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 becomes control dependent upon the statement that branches. All the statements in the two branches of the conditional now becomes control dependent upon their respective region nodes. This is illustrated in Figure 4.3. The attributes of statements stored in the statement table are defined below:

1. *Statement Type* indicates the type of the statement such as `call`, `if-then-else`, `while`, `for` etc.

2. *Statement Dependence Nesting Level* in the statement table is defined as the number of region nodes on the path from the root to it.

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

4. *Facility Used* is the set of facilities used by the statement.

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

6. *Childs* point to the statement table of the children (*left child or right child*), statements of the statement. This occurs when the statement happens to be an `if`, `while` or a `do` for which there are control dependences. For all other

```
S1      accessX(C1, X1);
S2      i.increment(X1);
S3      accessY(C1, Y2);
S4      Y2 := in.integer_copy(Y1);
S5      if i.equal(X1, X2) then
S6          accessX(C2, X2);
S7          if in.equal(Y1, Y2) then
S8              re.rec1_access(C1, X2);
S9              re.rec2_access(C2, Y1);
            else
S10             re.rec1_access(C2, X1);
S11             re.rec2_access(C1, Y2);
            end if
S12         Y1 := in.gen_five;
S13         return true;
        else
S14         re.rec1_access(C1, X1);
S15         return false;
        end if
```

Figure 4.4 The *coordsEqual* operation of *coordinate* module

statements, this would be a *null* pointer. Thus the statement table graph is a binary tree with each statement having a left child or a right child depending upon its *statement type*.

The algorithm for building the CDG from the statement table is shown in Figure 4.7. We select the *coordsEqual* operation of the *coordinate* module (the *vehicle application*, explained in Chapter 2), as an example program segment to illustrate all the dependence graphs and the cloning analysis thereafter. The *coordsEqual* operation is shown in Figure 4.4. The CDG of the operation is shown in Figure 4.5 and the statement table of the component is illustrated in Figure 4.6. We now discuss the data dependence graph.

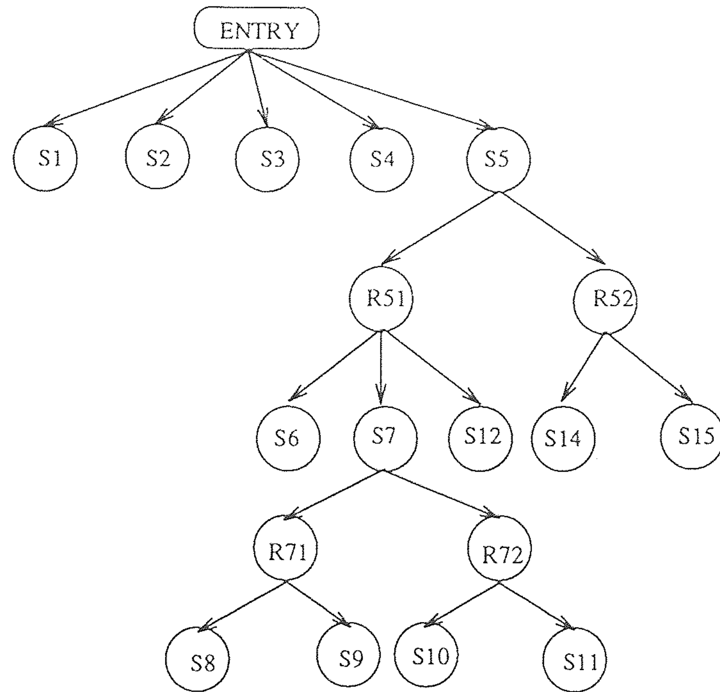


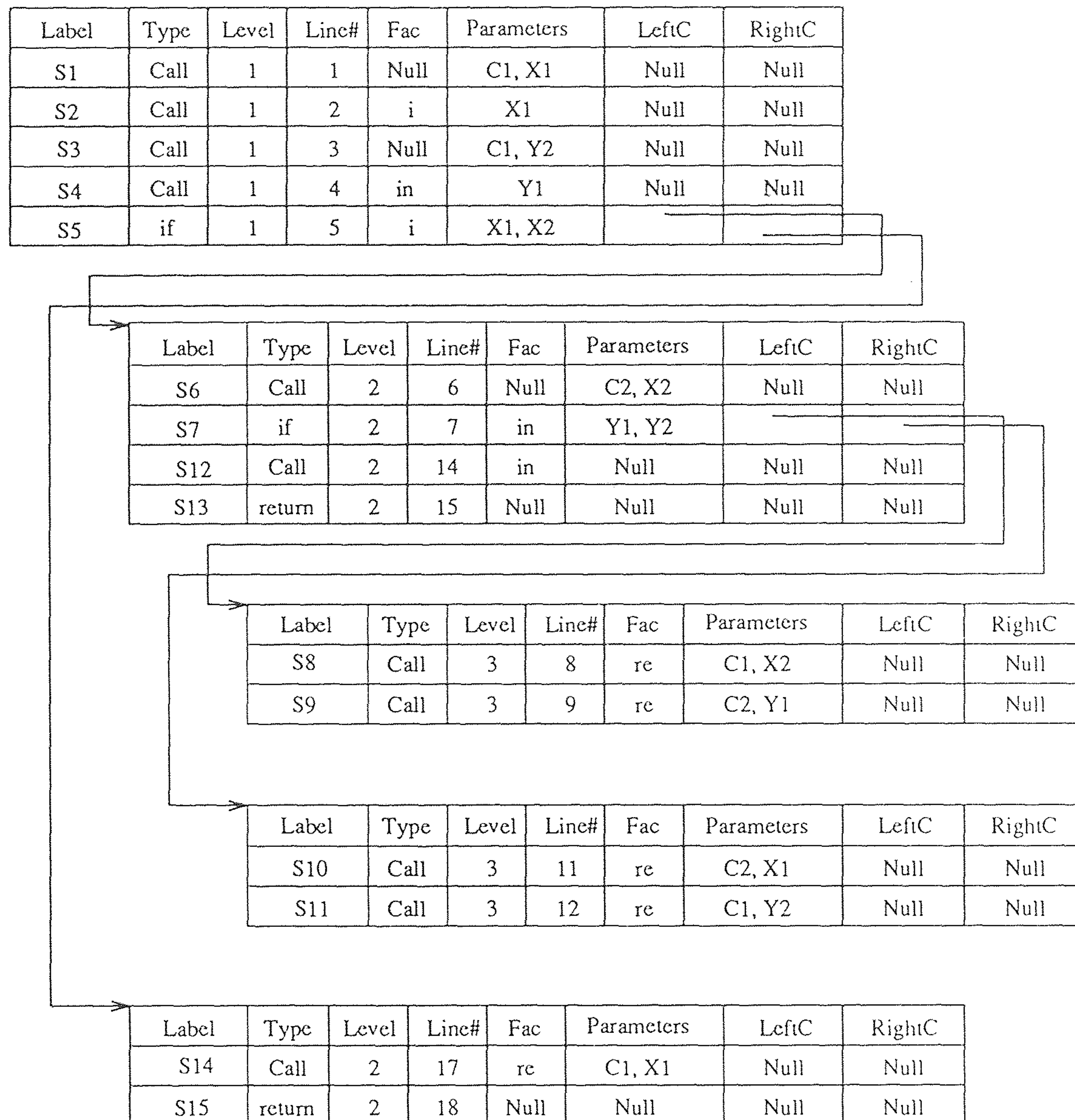
Figure 4.5 The CDG of *coordsEqual* operation

4.2.1.2 Data Dependence Graph *For any two statements S_i and S_j, if S_j is lexically after S_i and S_j needs some parameters which were passed on to S_i, then S_j is* **data dependent** *upon S_i.*

Intuitively, this means that the statement S_j must wait for the completion of statement S_i in order to access the data used by S_i. This data dependence between S_i and S_j is denoted as $S_i \rightarrow^d S_j$. Formally, the Data Dependence Graph is defined as follows.

$$DDG(S_i, S_j) = \begin{cases} 1, & \text{if } S_i \rightarrow^d S_j; \\ 0, & \text{otherwise;} \end{cases}$$

We now present the algorithms for building the DDG. The main algorithm is shown in Figure 4.8, and the supplementary ones in Figure 4.9 & Figure 4.10. The DDG of the *coordsEqual* operation (Figure 4.4) obtained by applying the algorithms, is shown in Figure 4.11.



Label	Type	Level	Line#	Fac	Parameters	LeftC	RightC
S1	Call	1	1	Null	C1, X1	Null	Null
S2	Call	1	2	i	X1	Null	Null
S3	Call	1	3	Null	C1, Y2	Null	Null
S4	Call	1	4	in	Y1	Null	Null
S5	if	1	5	i	X1, X2		

Label	Type	Level	Line#	Fac	Parameters	LeftC	RightC
S6	Call	2	6	Null	C2, X2	Null	Null
S7	if	2	7	in	Y1, Y2		
S12	Call	2	14	in	Null	Null	Null
S13	return	2	15	Null	Null	Null	Null

Label	Type	Level	Line#	Fac	Parameters	LeftC	RightC
S8	Call	3	8	re	C1, X2	Null	Null
S9	Call	3	9	re	C2, Y1	Null	Null

Label	Type	Level	Line#	Fac	Parameters	LeftC	RightC
S10	Call	3	11	re	C2, X1	Null	Null
S11	Call	3	12	re	C1, Y2	Null	Null

Label	Type	Level	Line#	Fac	Parameters	LeftC	RightC
S14	Call	2	17	re	C1, X1	Null	Null
S15	return	2	18	Null	Null	Null	Null

Figure 4.6 The Statement Table of *coordsEqual* operation

```
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 4.7 Algorithm for building CDG

```
SearchDD(tt : StatementType)
  PS : stack(StatementType)
  begin
    if (tt.rightc ≠ null) or (tt.leftc ≠ null) then
      Push tt.rightc & tt.leftc into stack PS;
    else
      begin
        st = successiveStatement(tt);
        if (st ≠ null) then stack.push(st, PS);
      end
    while not stack.empty(PS) do
      begin
        st = stack.pop(PS);
        if (st ≠ null) then
          begin
            if (checkDD(st,tt) = true) then
              begin
                DDG(tt,st) = true;
                Remove (st.Parameters ∩ 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 ≠ null) then stack.push(st, PS);
                    else flag = true;
                  end
              end
            else
              if (st.rightc ≠ null) or (tt.leftc ≠ null) then
                Push st.rightc & st.leftc into stack PS;
              else
                begin
                  st = successiveStatement(tt);
                  if (st ≠ 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 4.8 Algorithm for building DDG *(searching for data dependence)*

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

Figure 4.9 Algorithm for building DDG *(finding successive statement)*

```
checkDD(st, tt : StatementType) : boolean;
  begin
    if (st.Parameters ∩ tt.Parameters = ∅) then
      return true;
    else
      return false;
  end checkDD
```

Figure 4.10 Algorithm for building DDG *(finding common parameters)*

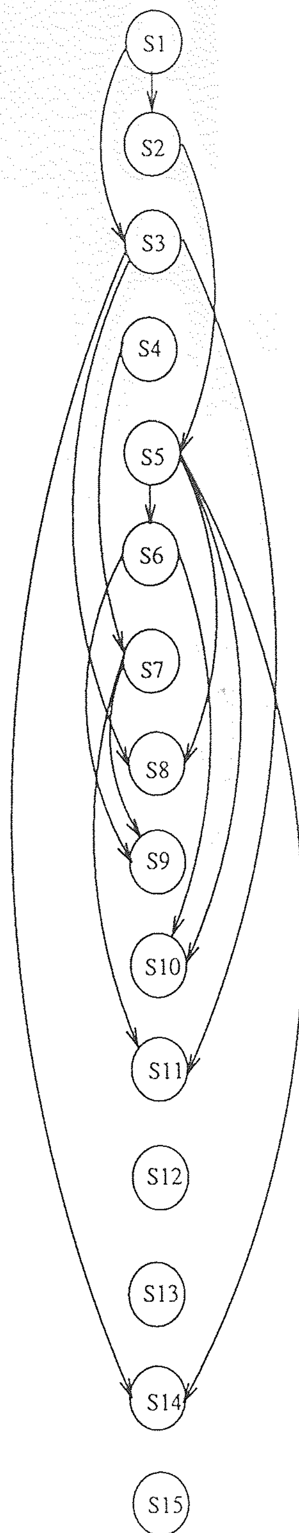


Figure 4.11 The DDG of *coordsEqual* operation

```
BuildCDDG(DDG,CDG,CDDG)
  begin
    copy CDG to CDDG;
    for each S_i →^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 BuildCDDG
```

Figure 4.12 Algorithm for building CDDG

4.2.1.3 Program Dependence Graph & Facility Dependence Graph The data dependences represented in the DDG could be built into the CDG, and a new graph called the *Control and Data Dependence Graph (CDDG)* could be formed. This graph represents the combination of control and data dependences between the program statements. The algorithm for building the CDDG from the DDG and the CDG is shown in Figure 4.12.

The CDDG also represents parallelism relationship between the statements. Any two statement nodes in the CDDG, could run in parallel if they do not have any transitive closed dependence relations. That is, statements which are dependent on one another (either through control or through data and by direct dependence or by ancestral dependence) cannot execute concurrently.

In all the graphs discussed so far, the possible code contention or facility dependence between the program statements have not been considered. Facility dependence between statements is defined as:

For any two statements S_i and S_j that use the same facility, if S_j is lexically after S_i, then S_j is said to be **facility dependent** *upon S_i.*

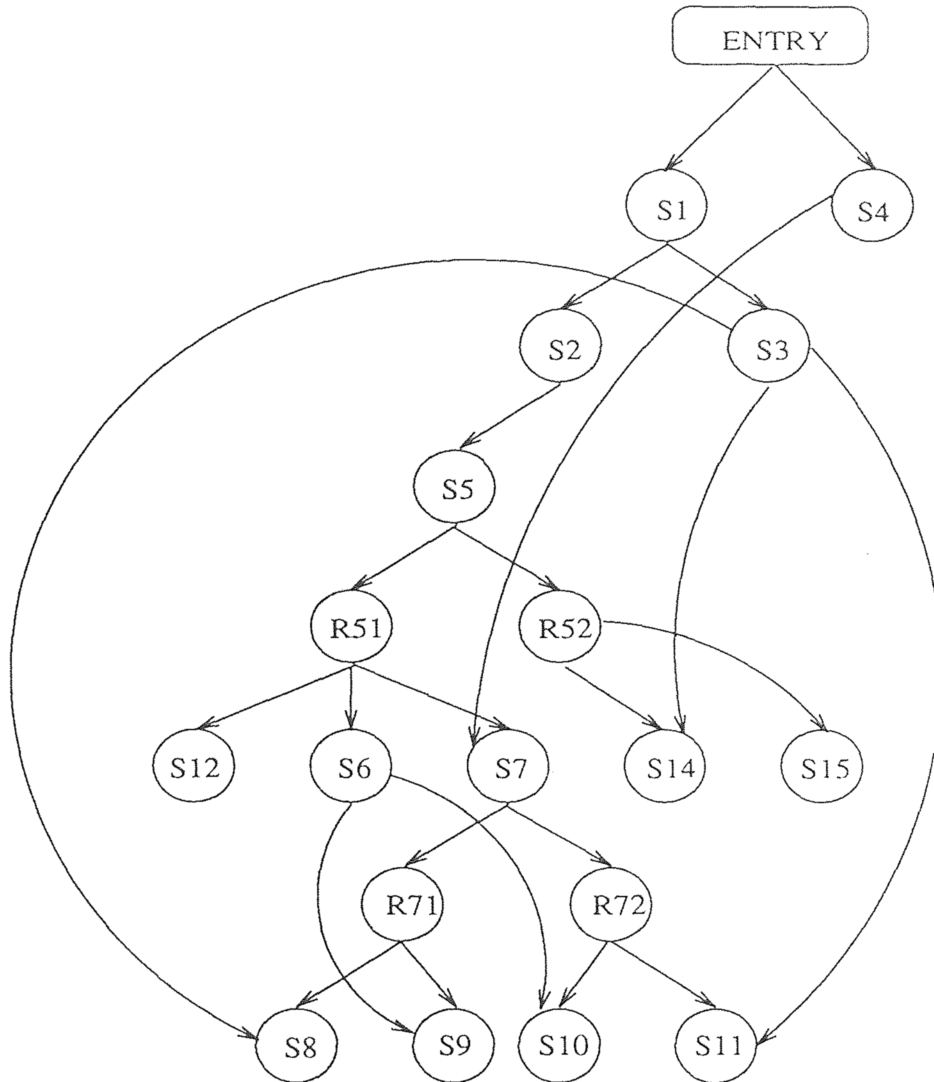


Figure 4.13 The CDDG of *coordsEqual* operation

Formally the Facility Dependence Graph (FDG) is,

$$FDG(S_i, S_j) = \begin{cases} 1, & \text{if } S_i \rightarrow^f S_j; \\ 0, & \text{otherwise;} \end{cases}$$

By adding facility dependence into the CDDG, the graph consists of three kinds of dependences - control, data, and facility. We call the new graph as the **Program Dependence Graph (PDG)**. The CDDG of the *coordsEqual* operation is illustrated in Figure 4.13 and the PDG, in Figure 4.14. The FDG of the operation, indicating only the facility dependences is shown in Figure 4.15.

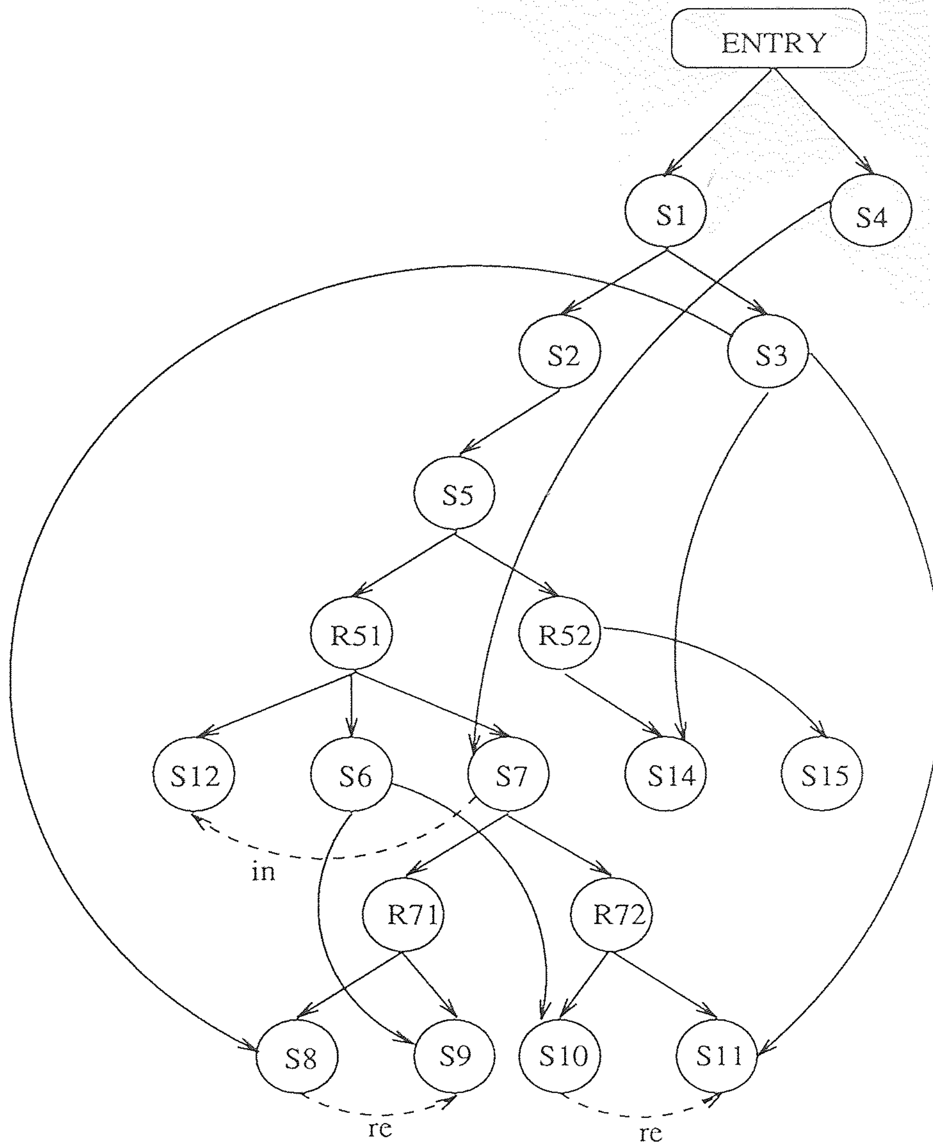


Figure 4.14 The PDG of *coordsEqual* operation

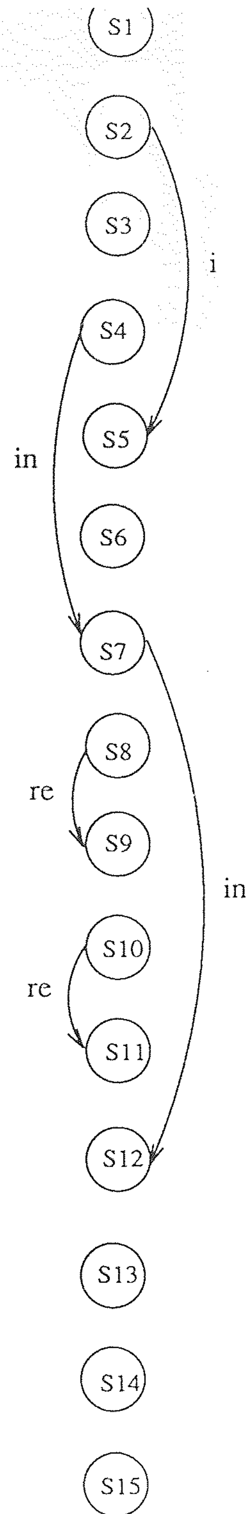


Figure 4.15 The FDG of *coordsEqual* operation

Note that while building the PDG, we add only those facility dependences into the CDDG (from FDG) that connects a node with one of its siblings. In other words, we do not add any facility dependence that connects a node to its descendant in the PDG. This is because, the facility dependences represented in the FDG may be affected by data and control dependences. Therefore, even if we have nodes depending on the same facility in the FDG, the inherent potential parallelism (the idea being that every edge in the FDG, represents code contention, and that could be removed by cloning the code, thus enhancing concurrency) is made ineffective by the presence of a data or control dependence, which cannot be removed at any cost.

4.2.2 Extracting Parallelism from Graphs

Identifying program statements that contend for a facility is accomplished by considering the DDG, CDG and FDG in conjunction. Extracting the cloning requirements of facilities considering all the graphs simultaneously have been discussed in [1]. In [1] by Welch, the idea is to cluster program statements in an operation (a method in the module) which due to the data dependences among them has to execute in order, into what is called *units*. The statements of a unit therefore cannot contend for a facility but different units in an operation may, with each other. Also, each unit can utilize only one clone of each of the facilities that it uses, since the statements of a unit must execute sequentially. Algorithms have been proposed by Welch for identifying the units and thereafter for grouping those units which could be run in parallel. A group therefore would then contain a set of units in which, each unit can run in parallel with the every other. Given the groups, a *Group Facility Matrix* is then constructed to determine the number of clones of a facility that can be used concurrently. Each row of this matrix corresponds to a group, each column a facility, and each entry indicates the number of clones of the facility needed by the group. The maximum number in a column of the matrix is an

upper bound on the number of clones of a facility that can be used simultaneously. The results include transformation rules for conditionals, so that the clone analysis algorithm avoids considering an exponential number of paths through the program. Also, techniques have been proposed by Welch to determine the number of clones required each time a loop is unrolled.

In this work, the dependence graphs extracted from each operation of a module is subjected to the cloning analysis algorithms of Welch, for determining an upper bound on the facility clone requirements.

4.2.3 Cloning Analysis of the Application Program

In this section, we illustrate how the cloning analysis techniques are applied to the graphs for revealing concurrency and to further drive home the idea. the algorithms developed are applied to the application program and the results are shown.

The Data Dependence Graph extracted for the *coordsEqual* operation shown in Figure 4.11 illustrates, constraints to the ARPC model due to data at the statement level. Dependences due to data, though poses threat to concurrency, have to obeyed strictly to maintain program correctness and is done so in this work. However, a set or collection of program statements having heavy data dependences among them could be identified from the code and could be executed concurrently with other similar sets if any, provided these sets between them do not have any dependences. Welch [1], proposes theorems in this regard for identifying such collection of statements (called *units*) in the program. The algorithm developed for extracting *units* from the program is shown in Figure 4.16 and Figure 4.17.

Once clusters of program statements having data dependences have been identified, the *Units Parallelism Matrix, (UPM)* is constructed which shows the potential concurrency in the program at the unit level. This matrix defines parallelism relation between units or in other words, indicates which units could run in

```
GetUnits(DDG, UNITS)
  var Q : QUEUE of DDG_node type;
  begin
    insert_node(UNITS); /* inserts a new node in UNITS */
    for each node in DDG do
      begin
        while not EndNode(node,DDG) do
          begin
            if (node ≠ null) then
              node=next_stmt(node,DDG,Q,UNITS);
            else
              break;
            insert_stmt(UNITS,node); /* inserts the graph stmt into the UNITS */
          end while
        if (node ≠ null) then
          node=next_stmt(node,DDG,Q,UNITS);
        else
          break;
        insert_node(UNITS);
        insert_stmt(UNITS,node);
      end for
  end GetUnits
```

Figure 4.16 Algorithm for Finding Units (*main*)

```
next_stmt(node : DDG_node, graph_node : DDG_node, Q : QUEUE, Units : UNITS)
                                    returns DDG_nodetype;
  var x : DDG_nodetype;
  begin
    while not graph_node ≠ null do
      /* i.e, for each node in DDG */
      begin
        if (graph_node.label = node.label) then
          begin
            if (graph_node.next ≠ null) then
              if ¬(MergeNode(graph_node.next)) or
                          ∀_{S∈immediate_predecessors(graph_node.next)}(S ∈ Units) then
                ENQUEUE(graph_node.next,Q);
          end
        graph_node = graph_node.next;
      end while
    if (EndNode(node) = true) then
      begin
        x := FRONT(Q);
        DEQUEUE(Q);
        if (x ∩ Units = ∅) then
          return x;
      end
    else
      begin
        x := dependent_stmt(node,DDG); /* the statment dependent upon node */
        remove_stmtQ(x,Q); /* remove the statment x from the QUEUE */
        return x;
      end
    if (∃_{S∈DDG}(S ∩ Units = ∅)) then
      return S;
  end next_stmt
```

Figure 4.17 Algorithm for Finding Units (*finding next statement*)

parallel with each other. Based on the UPM, we then construct groups of units in which every unit can run in parallel with every other. The algorithm for building parallel units is shown in Figure 4.18.

The basic facility requirements of the units is illustrated in the *Facility Unit Matrix (FUM)*. This matrix has all the facilities used in the program as its row elements and the units identified, along its column. Note that no unit can have more than a single requirement of a facility in FUM, even if that unit incorporates statements using the same facilities. This is because, the statements have been clustered together to form a unit since they have data dependences in the first place and therefore such a facility dependence is totally *ineffective*. *Units* thus represents the basic units of parallelism in this work. An exception to the above stated fact (regarding FUM) occurs when conditional statements appear in the program and this will be discussed subsequently. We use FUM later on to build the Group Facility Matrix (GFM), which finally shows the upper bound on the cloning requirements of the facilities used.

4.2.3.1 Conditional Handling We continue to use the *coordsEqual* operation of the *vehicle* module which has been used throughout this work as the application example, for illustrating the cloning analysis also. The *coordsEqual* operation (Figure 4.4) has conditional statements in its code, and this calls for applying the transformation algorithms first, before it could be subjected to a complete clone analysis. The transformation algorithms causes the DDG of the code to be metamorphosed into a graph where all the conditional statement nodes (the statements appearing inside the body of the conditional) are replaced with a single node (a super node) having specific cloning needs. The idea of transforming the conditional statements in the graph is to avoid considering an exponential number of paths through the program for determining an upper bound on the clone requirements.

```
BuildGroups(UPM,GROUPS)
  begin
    num_groups := 0; /* total number of groups */
    /* For each row of P, i.e., for each unit i, */
    /* Build groups containing i and units parallel to i. */
    for i := 1 to NUM_UNITS in UPM do
      begin
        /* Create a group containing only i. */
        num_groups++;
        start := num_groups;
        end := num_groups;
        GROUPS(num_groups) := {i};
        for j := i+1 to NUM_UNITS do
          begin
            /* For each column of P, i.e., for each unit j. */
            if (P(i,j) = 1) then
              stop := false;
              for k := start to end do
                begin
                  /* does j fit into an existing group? */
                  if ∀_{u∈groups(k)}(P(j,u) = 1) then
                    begin
                      GROUPS(k) := groups(k) ∪ {j}
                      stop := true;
                    end if
                end for
              if (stop = false) then
                begin
                  /* Make a new group for i,j */
                  end++;
                  num_groups++;
                  GROUPS(end) := {i,j};
                end if
            end if
          end for
      end for
  end BuildGroups
```

Figure 4.18 Algorithm for grouping Parallel Units

```
ConditionalTransform(DDG,StaTab)
  begin
    for each entry N in StaTab do
      begin
        if (N.type = "if") then
          Depth_First_Search(N,DDG);
      end for
  end ConditionalTransform
```

Figure 4.19 Algorithm for Transforming Conditionals (*main*)

For the transformation, we first identify the boundaries of the conditional in the DDG and then replace every edge crossing the boundaries (from the outside of the conditional body) with edges to the boundary. For example, a directed edge (P,Q) crossing the beginning of the conditional (P preceding the conditional, and Q within the conditional), is replaced with edges (P,C) and (C,Q), where C is the start of the conditional. Edges crossing the end limits of the conditional body are transformed in a similar way. Once transformed, the graph of the conditional body. which is now totally independent with respect to the outside program domain is extracted out. The extracted DDG is then subjected to the cloning algorithms discussed in Figure 4.16, Figure 4.17 and Figure 4.18, considering each branch of the conditional in isolation with the rest. Units, Groups, UPM, FUM, and GFM are formed for each branch and the maximum of the clone requirements (of facilities per group) of all the branches is determined. This maximum value represents an upper limit on the cloning needs of the entire conditional. The conditional is then defined in the DDG as a single node with these specific cloning needs. The DDG is thus transformed into a graph defining a single thread of execution.

While constructing the Units, Groups etc., for the single scenario DDG, we consider the transformed super node like any other program node. However. when the Facility Units Matrix (FUM) is constructed, the facility requirements of the

```
Depth_First_Search(N : StaTab, DDG)
  begin
    for each childStaTab C of N in StaTab do
    /* childStaTab is either C.LeftC or C.RightC */
      begin
        for each entry x in C do
          begin
            if (x.type = "if") then
              Depth_First_Search(x,DDG);
          end for
      end for
    Transform_Edges(DDG,N);
  end Depth_First_Search
```

Figure 4.20 Algorithm for Transforming Conditionals (*Depth-First-Search*)

super node would be that of the previously determined one. Nested conditionals are handled by transforming them inside-out. That is, the conditional nested at the deepest level say n, is transformed first. Then, it is treated as an atomic unit while the conditional at level n - 1 is transformed. The transformation continues at successively shallower levels of nesting, until all conditionals are transformed. We perform a *Depth-First-Search* on the graph for such a transformation. The algorithm for transforming the conditionals is shown in Figure 4.19 and Figure 4.20. Note that the algorithm needs the statement table also, since all information regarding the program structure is stored in it, where as the DDG reveals only the data dependence relations.

Since the DDG of the *coordsEqual operation* contains conditional statements (nesting at 2 levels), it is first *filtered* through the transformation algorithms. The transformation at level 2 is shown in Figure 4.21. The four different graphs in the Figure 4.21, illustrate the transformation process of the DDG. The initial DDG shown at the left extreme (same as that in Figure 4.11) is the untransformed graph showing all the data dependences between the program statements ignoring their

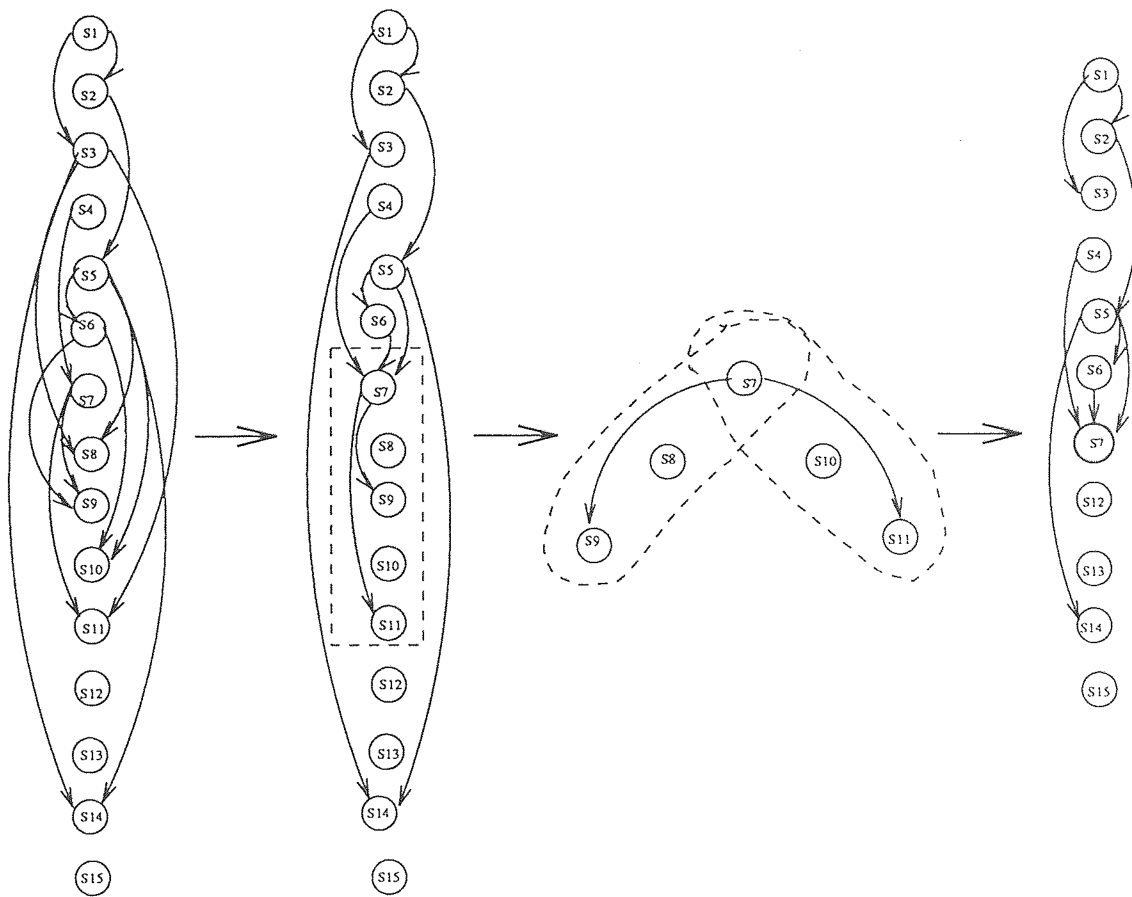


Figure 4.21 Conditional Transformation of *coordsEqual* operation at level 2

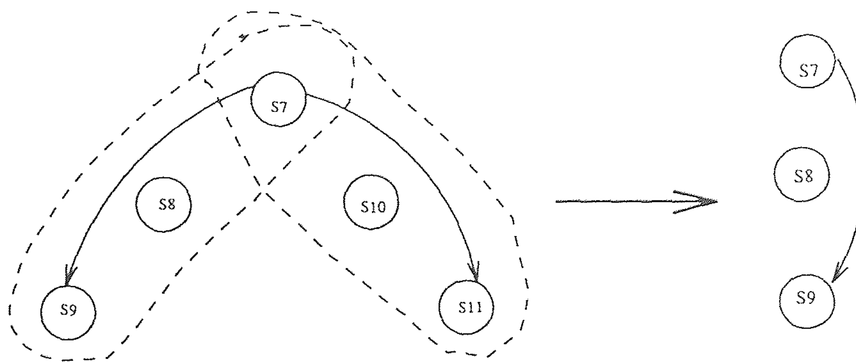


Figure 4.22 Extracting Left Graph of Conditional (level 2)

control dependences. Note that the statements *S8*, *S9 S10* & *S11* belong to the innermost conditional (at level 2) and therefore, have a control dependence upon the statement *S7* with the same precedence. Once these set of statements are identified from the graph, the next step is to transform the edges crossing the conditional boundaries (*S7* - *S11*) as discussed before. The conditional statement nodes now become *data independent* with the outside program statements and it is extracted out. The extracted graph is then split into different graphs simulating all the possible execution paths in the program. Thus we have as many graphs as the possible run time scenarios. The programming model supports only the *if* statement as a conditional construct and therefore the *splitting* (of graphs) is always limited to (a maximum of) two - the *left graph* and the *right graph*.

Each of the graphs (*left* and *right* graphs) is then subjected to the cloning analysis algorithms separately. The extracted *left graph* (at level 2) is shown in Figure 4.22.

Statements *S7*, *S8* and *S9* forms the *left* graph. *Units* are then identified and the *Units Parallelism Matrix (UPM)* is constructed. This is followed by the grouping of parallel *Units* and the construction of the matrices, *Facility Unit Matrix (FUM)* & *Group Facility Matrix (GFM)*. We illustrate the *Units*, *Groups* and all the other matrices in Appendix D. The last row of the matrix *GFM*, indicates the maximum number of clones of the facilities *i*, *in* and *re* that could be used inside the inner condi-

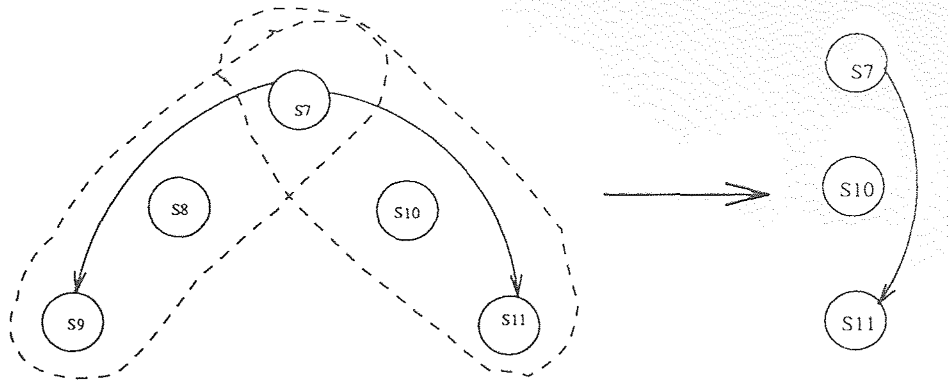


Figure 4.23 Extracting Right Graph of Conditional (level 2)

Left Graph Clone Requirements

i	in	re
0	1	2

Right Graph Clone Requirements

i	in	re
0	1	2

Max Clone Requirements

i	in	re
0	1	2

Figure 4.24 Cloning Requirements of Conditional (at level 2)

tional, if the execution of the *left* graph (at level 2) occurs at run time. Statements *S7*, *S10*, *S11* forms the *right* branch of the conditional at level 2. The *right* graph is therefore constructed with these statements and is shown in Figure 4.23. Note that the conditional evaluation statement *S7*, forms part of both the *right* and *left* graphs when considered for the cloning analysis, since we are trying to speculate the possible execution scenarios. The construction of *Units*, *Units Parallelism Matrix (UPM)*, *Groups*, *FUM* and *GFM*, then proceeds in the same way as before. All these matrices are illustrated in Appendix D.

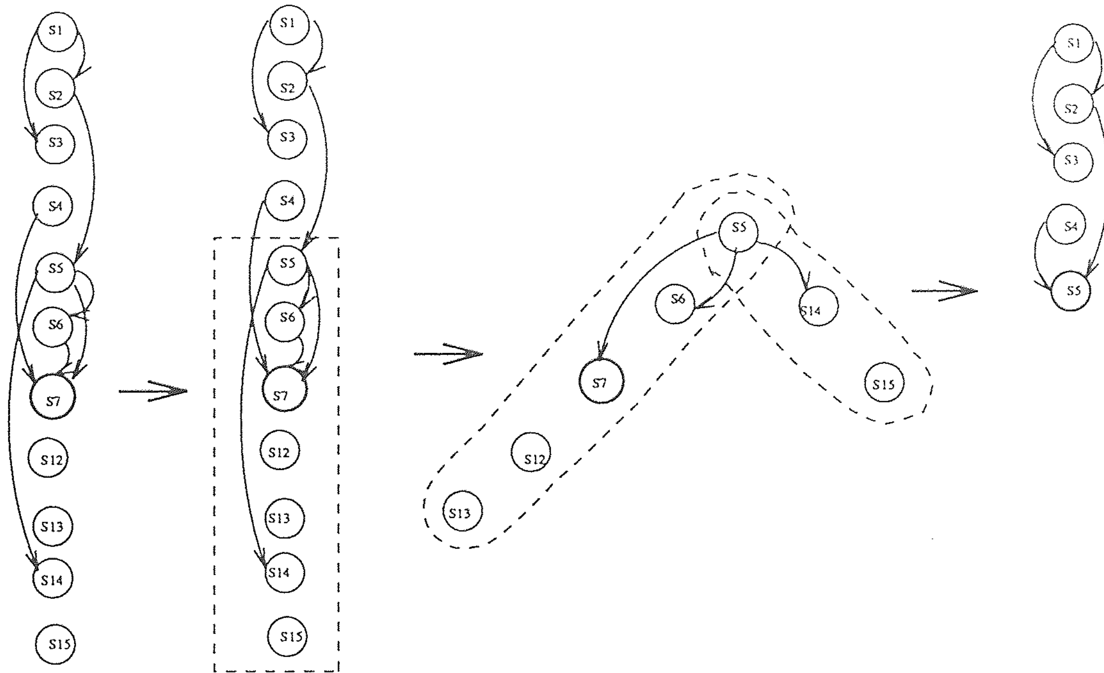


Figure 4.25 Conditional Transformation of *coordsEqual* operation at level 1

It is found that *0* clones of *i*, *1* clone of *in* and *2* clones of *re* are required for both the *left* and *right* branches of the conditional. Maximizing the cloning needs for the two cases though does not make any difference, is still shown in Figure 4.24 to illustrate the algorithm. Once the upper bound on the cloning needs of the (innermost) conditional have been determined, we now replace the entire conditional body (*statements S7 - S11*) with a single node (i.e., *S7*) in the DDG. The cloning needs of the statement node *S7* (i.e., *0* of *i*, *1* of *in* and *2 of re*) is recorded separately. The transformed DDG is shown in Figure 4.25. Now we transform the conditional at level 1, considering the conditional at level 2 as a single node, *S7*. The conditional body (at level 1) is then identified (statements *S5, S6, S7, S12, S13, S14* & *S15*) and the edges are transformed like before. The process is shown in Figure 4.25. The *left* and *right* graphs are extracted out from the transformed DDG. The *left* graph (statements *S5, S6, S7, S12* & *S13*) is shown in Figure 4.26. Note that, the statement *S7* which is a *supernode*, becomes *Unit 3* during the analysis and in the *Facility Units Matrix (FUM)*, its cloning needs have been assigned as the

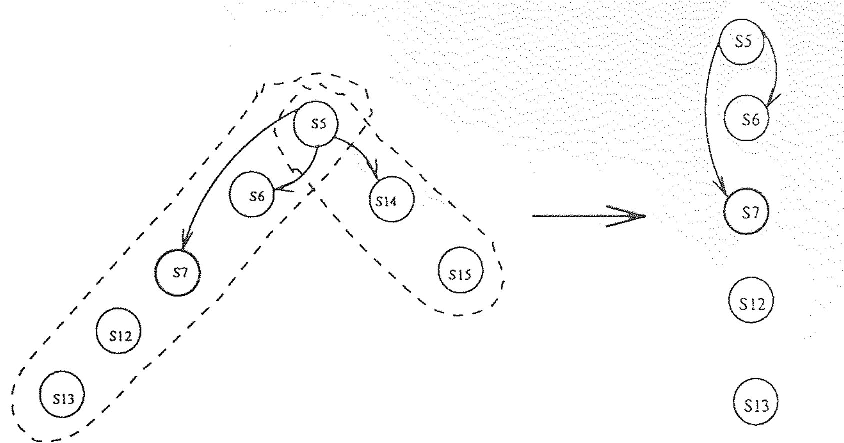


Figure 4.26 Extracting Left Graph of Conditional (level 1)

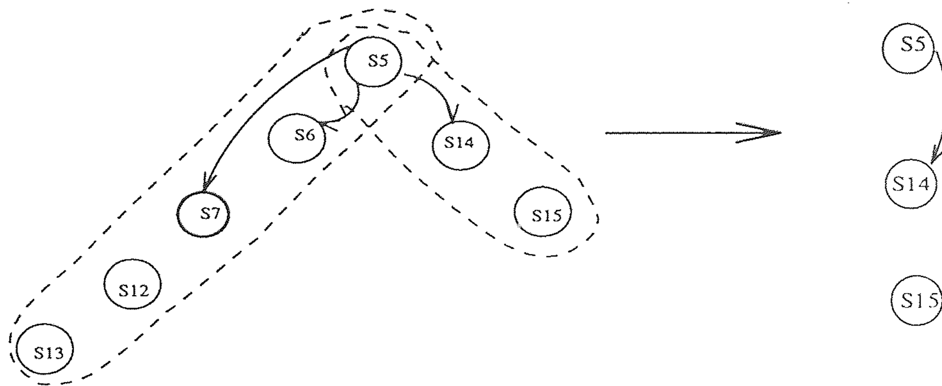


Figure 4.27 Extracting Right Graph of Conditional (level 1)

predetermined clone requirements of the entire conditional statements at level 2. The *Units*, *Groups* and the other matrices constructed are shown in Appendix D. The *right* graph (statements *S5, S14* & *S15*) is shown in Figure 4.27. The different matrices generated during these transformations are also shown in Appendix D.

Maximizing the cloning needs of the *left* and *right* graphs gives us an upper bound on the cloning requirements of the conditional at level 1. This is illustrated in Figure 4.28. Finally, the entire conditional statements (*S5* - *S15*) is replaced in the DDG with the supernode *S5*. We show the final transformed DDG in Figure 4.29.

Left Graph Clone Requirements

i	in	re
1	2	2

Right Graph Clone Requirements

i	in	re
1	0	1

Max Clone Requirements

i	in	re
1	2	2

Figure 4.28 Cloning Requirements of Conditional (at level 1)

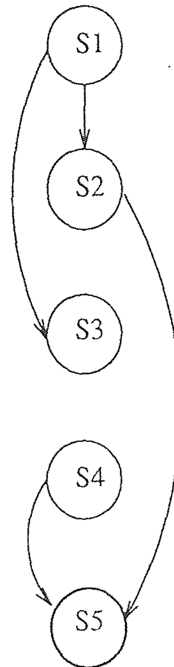


Figure 4.29 Transformed DDG of *coordsEqual* operation

Table 4.1 Units of *coordsEqual* operation

_	_
Un1	S1
Un2	S2
Un3	S3
Un4	S4
Un5	S5

Table 4.2 Units Parallelism Matrix of *coordsEqual* operation

_	Un1	Un2	Un3	Un4	Un5
Un1	_	0	0	1	0
Un2		_	1	1	0
Un3			_	1	0
Un4				_	0
Un5					_

4.2.3.2 Clone Analysis of Transformed DDG Units formed from the transformed DDG in Figure 4.29 by applying the algorithm in Figure 4.16 and Figure 4.17 is shown in Table 4.1. Note that the statement *S5* is a super-super node representing two nested conditionals. The Units Parallelism Matrix showing the parallelism relation between the units is illustrated in Table 4.2. Groups formed from the Units in Table 4.1 by applying the algorithm discussed in Figure 4.18 is shown in Table 4.3.

Table 4.3 Groups of Units of *coordsEqual* operation

_	_	_	_
Gr1	Un1	Un4	
Gr2	Un2	Un3	Un4
Gr3	Un3	Un4	
Gr4	Un4		
Gr5	Un5		

Table 4.4 Facility Units Matrix of *coordsEqual* operation

_	Un1	Un2	Un3	Un4	Un5
i		1			1
in				1	2
re					2

Table 4.5 Group Facility Matrix of *coordsEqual* operation

_	i	in	re
Gr1	0	1	1
Gr2	1	1	1
Gr3	0	1	0
Gr4	0	1	0
Gr5	1	2	2
Max	1	2	2

The Facility Units Matrix illustrating the facility requirements of the different units is shown in Table 4.4. Notice that the facility needs of Unit *5* was predetermined and it represents the requirements of the nested conditional statements.

Finally, the Group Facility Matrix is constructed which illustrates the facility requirements per group. The matrix is shown in Table 4.4. The last row in the matrix indicates the maximum number of clones of the facilities in *coordsEqual* operation that could used concurrently per group, and it represents an upper bound on the cloning requirements.

4.2.4 Parallelism inside Loops

The opportunities for parallelism that exist inside loops (both bounded and unbounded) through clones, could be identified by unrolling them. Unrolling a loop simply means extending the code of the loop beyond a single iteration. The idea of unrolling a loop is to reveal chances of parallelism between loop iterations which, due to facility dependences (across iterations) may be getting lost. Removing

```
procedure AccessSeq
  parameters
    preserves comp: pcompType;
    produces seq: seqType;
  end parameters

  begin
    local variables
      temp : op.optype;
      diff : in.integer;
      newq : q.queue;
    end local variables

      diff := in.gen_five;
      q.SetSize(seq,diff);
      while not q.IsEmpty(newq) do
        q.dequeue(newq,temp);
        q.enqueue(seq,temp);
      end while;
  end AccessSeq
```

Figure 4.30 The procedure *AccessSeq* of module *pcomp*

these dependences by providing additional clones of the facilities causing it, we enhance concurrency inside loops, further rendering accuracy to the clone analysis. We illustrate this process of exploiting parallelism inside loops with an example operation (*procedure AccessSeq*) shown in Figure 4.30. The *procedure AccessSeq* is actually defined in the interface section of the module *pcomp* used in the *vehicle* application. The complete module is given in Appendix C.

The procedure *AccessSeq* incorporates a simple loop mechanism. We use this unbounded loop (the *while*) to illustrate the clone analysis of loops. The DDG of the procedure *AccessSeq* is shown in Figure 4.31. For the purpose of clone analysis, while constructing the DDG, we ignore the presence of loops and treat them as mere straight line code. The DDG of the loop (extracted out from the rest of the graph) is shown in Figure 4.32.

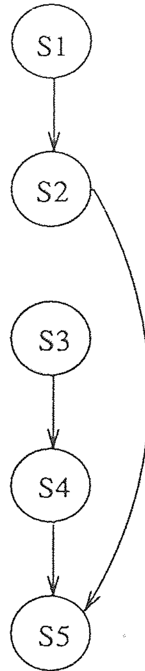


Figure 4.31 DDG of operation *AccessSeq* (module *pcomp*)

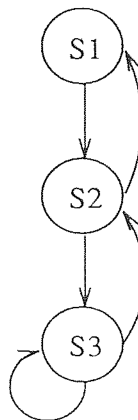


Figure 4.32 DDG of the Loop (operation *AccessSeq*)

Note that the graph contains backward edges indicating cross iteration dependences i.e., statements from one iteration of the loop depending on statements from others. Now if the loop is unrolled once (i.e., considering two iteration executions of the loop), the backward dependences would appear as forward dependences. We show the DDG of the once unrolled loop in Figure 4.33. The facility dependences between statements across iterations are then added into the unrolled loop DDG to reveal parallelism between loop iterations. This graph is shown in Figure 4.34. There exists a pure facility dependence between statements *S3* and *S4* due to contention for the facility q. An additional clone of q can resolve this contention and thereby statements *S3* and *S4* can be executed concurrently. Such a potential concurrency and thereby the additional clone requirement of the facility q is revealed only after unrolling the loop and this justifies the overhead of such an analysis at compile-time or even at link-time. Thus the total number of clones required for exploiting parallelism between all possible iterations of the loop is revealed by unrolling the loop as many times. But in general, unrolling the loop once is sufficient enough to determine the additional amount of clones required.

Also, there could be antidependences [12] between statements across loop iterations. In the loop DDG shown in Figure 4.34, the dependence between statements *S3* and *S5* due to the common parameter *temp* is an *antidependence*. Such an antidependence can be revealed at link-time by checking the parameter passing modes of the operations *enqueue* of statement *S3* and *dequeue* of statement *S5* (in module *queue*, q being an instance of it) and thereafter, could be removed by replacing the data (causing the antidependence) with a temporary variable, without affecting code correctness. Welch illustrates this aspect in [1]. In the illustrated application example, such a removal doesn't make any difference since the statements (*S3*, *S5*) do not have a facility dependence. We now propose an algorithm for determining the cloning requirements inside loops by the unrolling technique. The main

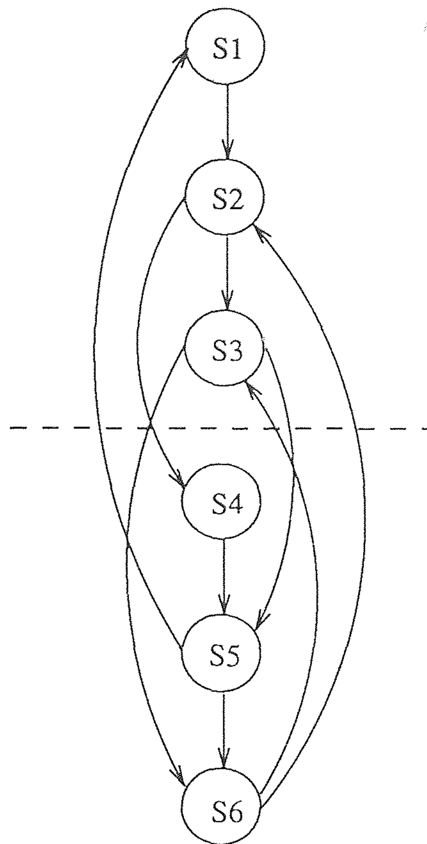


Figure 4.33 DDG of the Unrolled Loop

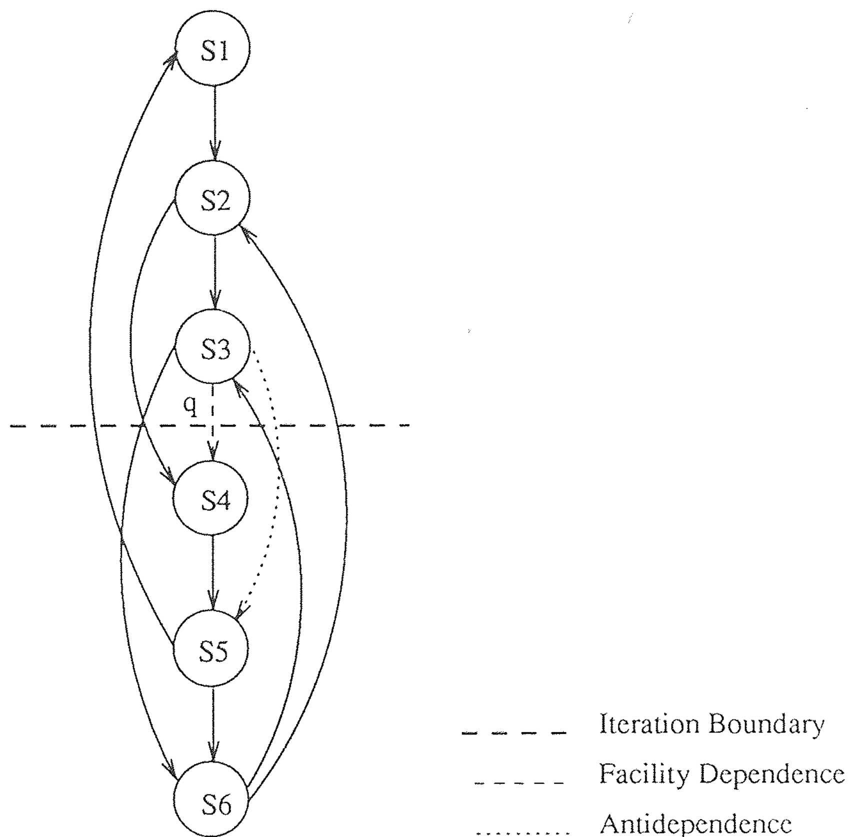


Figure 4.34 DDG of the Unrolled Loop with Facility & Antidependences

```
Loop_Handler(loopDDG, Fac_Clone_List)
  var x, y : DDG_nodetype;
    loopFDG : DDG type;
  begin
    Unroll_Loop(loopDDG);
    Remove_AntiDep(loopDDG);
    loopFDG := Cross_FacDepGraph(loopDDG);
    for each node x in loopDDG do
      begin
        for each node y in loopDDG do
          begin
            if (loopFDG(x,y) = true and loopDDG(x,y) = false) then
              Fac_Clone_List(x.fac,clones)++;
          end for
      end for
  end Loop_Handler
```

Figure 4.35 Algorithm for Handling Loops (*main*)

algorithm is given in Figure 4.35 and the supplementary ones are shown in Figure 4.36, Figure 4.37 and Figure 4.38.

4.2.5 Interfacility Clone Analysis

An application is developed in the proposed language model through separately written and compiled modules. Independent compilation of modules is a feature of our paradigm as discussed in Chapter 2. Until now, we have presented methods to compute cloning requirements of modules in an independent fashion. However, this framework needs to be extended when we have to deal with an entire application where modules are combined together and between which complicated call relationships often exist. In this section we discuss algorithms for computing the cloning requirements of the module instances used in an application based on their call relationships (Call DAG), as outlined by Welch in [1].

Interfacility clone analysis can be achieved by modifying the way in which the clone requirements of *units* were hitherto calculated. The facility cloning needs of

```
Unroll_Loop(loopDDG)
  var u, v : DDG_nodetype;
  begin
    copy loopDDG to tempDDG;
    for each vertex u in tempDDG do
      begin
        mark[u] := unvisited;
        loopDDG(u.iter) := 1;
    /* This is to distinct iterations */
      end for

    for each vertex u in tempDDG do
      if (mark[u] := unvisited) then
        begin
          mark[u] := visited;
          add a new node u in loopDDG;
          loopDDG(u.iter) := 2;
          for each v in tempDDG do
            if (tempDDG(u,v) = true) then
              begin
                mark[v] := visited;
                add a new node v in loopDDG;
                loopDDG(v.iter) := 2;
                add an edge from u to v in loopDDG;
              end if
        end if

      for each vertex u in loopDDG do
        SearchDD(u);
      /* This is to add the cross-iteration data dependences */
  end Unroll_Loop
```

Figure 4.36 Algorithm for Handling Loops (*Unrolling Loop*)

```
Remove_AntiDep(loopDDG)
  var u, v : DDG_nodetype;
    p : parameter type;
  begin
    for each vertex u in loopDDG do
      begin
        for each vertex v in loopDDG do
          begin
            if (loopDDG(u,v) = true and
                (loop(u.iter) := 1) & loop(v.iter) := 2)) then
              begin
                if (Read_Oper(u) and Write_Oper(v)) then
                  begin
                    remove the edge from u to v;
                    p := u.parameters ∩ v.parameters;
                    for each node n in loopDDG do
                      if (ancestor(n) = u) then
                        replace n.parameter with p;
                  end if
              end if
          end for
      end for
  end Remove_AntiDep
```

Figure 4.37 Algorithm for Handling Loops (*Removing Antidependences*)

```
Cross_FacDepGraph(loopDDG) returns DDG type;
  var u, v : DDG_nodetype;
    loopFDG : DDG type;
  begin
    for each vertex u in loopDDG do
      begin
        for each vertex v in loopDDG do
          begin
            if (loopDDG(u.iter) = 1 and loop(v.iter) = 2) then
              if (u.facility ∩ v.facility ≠ ∅) then
                loopFDG(u,v) = true;
          end for
      end for
      return loopFDG;
  end Cross_FacDepGraph
```

Figure 4.38 Algorithm for Handling Loops (*Adding Cross Iteration Fac. Dep's*)

a *unit* includes, the cloning requirements of the methods (of other facilities) called by the statements in the *unit*, in addition to the single facilities directly called by the statements. Welch [1] refers to this additional cloning needs of a statement (i.e., a method call in a *unit*) as its *Transitive Cloning Requirements* or *TCR*. For this purpose a function *clones(u,x)* is defined to denote the number of clones of facility x required by the unit u. The value of the function denotes the result of combining the direct and transitive requirements of u. Direct requirements of the unit is what the *Facility Unit Matrix* indicates and Transitive Cloning Requirements is determined by examining the needs of the methods invoked by the statements of the unit. Formally,

$$clones(s_i, x) = DCR(s_i, x) + TCR(s_i, x)$$

$$clones(u, x) = max_{s_i \in u}(clones(s_i, x))$$

The above definition is extended for groups and operations also. The cloning needs of a group g for a facility x is defined as:

$$clones(g, x) = sum_{u \in a}(clones(u, x))$$

```
DCR(s : statement_type, x : facility_type) returns clones : integer;
  var n : DDG_nodetype;
  begin
    for each node n in DDG do
      begin
        if (n.label = s.label) then
          begin
            if (n is a Supernode) then
              begin
                for each facility f in Fac_List of n do
                  if (f = x) then
                    return n.clones;
              end
            else
              if (n.facility = x) then
                return 1;
          end if
      end for
    return 0;
  end DCR
```

Figure 4.39 Algorithm for finding Direct-Clone-Requirements (*DCR*)

Similarly, the number of clones of facility x required by an operation op of a facility f is defined as:

$$clones(f.op, x) = max_{g \in f.op}(clones(g, x))$$

Computing the cloning requirements of an application begins with determining the direct cloning needs of each operation in a facility represented as a node in the Call DAG, starting at the root vertex of the DAG. The cloning needs are then re-computed using the above functions where we consider the transitive requirements also. We now present the algorithms for the interfacility clone analysis. The algorithm for computing *DCR* is shown in Figure 4.39, *TCR* in Figure 4.40 and finally, the clone needs for an entire application (*Program-Clone-Needs*) in Figure 4.41.

```
TCR(s : statement_type, x : facility_type) returns clones : integer;
  var clones : integer
    n : DDG_nodetype;
  begin
    if (s.facility = x) then
      return 1;
    else
      begin
        clones := 0;
        for each node n in DDG(s.facility,s.operation) do
          begin
            if (DAG(n.facility,x) = true) then
              clones := clones + TCR(n,x);
          end for
        clones := clones + GFM(s.facility,s.operation)(Max,x);
        return clones;
      end
  end TCR
```

Figure 4.40 Algorithm for finding Transitive-Clone-Requirements (TCR)

```
Program_Clone_Needs(DAG)
  var DDG, FDG, GFM : Graph_type;
    N : DAG_nodetype;
  begin
    for each node N in DAG do
      begin
        for each facility f used in N do
          for each operation O in N do
            Facility_Clones(f,clones) := Oper_CloneNeeds(f,GFM(N,O));
      end for
  end Program_Clone_Needs
```

Figure 4.41 Algorithm for finding Program-Clone-Needs

CHAPTER 5

ILLUSTRATION OF SYSTEM DESIGN

In this Chapter, we discuss the system design of the implementations of the compiler. Note that, all the parallelism information (i.e., graphs and cloning needs) is extracted from the source code at compile-time. The system design at the top most level is shown in Figure 5.1.

The compiler, while compiling the application source code, extracts the different dependence graphs and the facility-cloning needs from it. As discussed in Chapter 2, the modules are compiled separately and linked together by the linker, before being loaded. The graph extraction is done by the compiler after generating the intermediate representation i.e., the *Statement Table*, which is done while parsing the source program. The *Statement Table* thus becomes the direct output of the parser and is then given to the *Graph Extractor*. The *Graph Extractor* generates the different dependence graphs. The graphs are produced in the form of separate files, the naming convention of which was outlined in Chapter 2. The *Data Dependence Graph (DDG)* from the *Graph Extractor* is then *filtered* through the *Graph Filter*. This filtering process transforms the conditionals and handles the loops if any, in the *DDG*. The transformed *DDG* is then sent to the *Cloner* which then generates all the matrices required for the concurrency analysis. The final matrix *(Group Facility Matrix)* generated by the cloning routines becomes the end output of the compiler

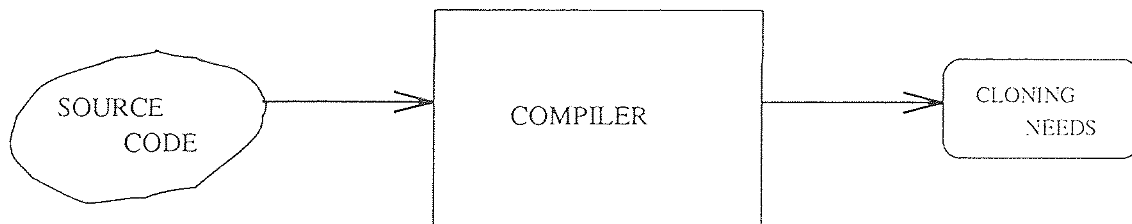


Figure 5.1 System Design of The Compiler at the *top level* or *level 1*

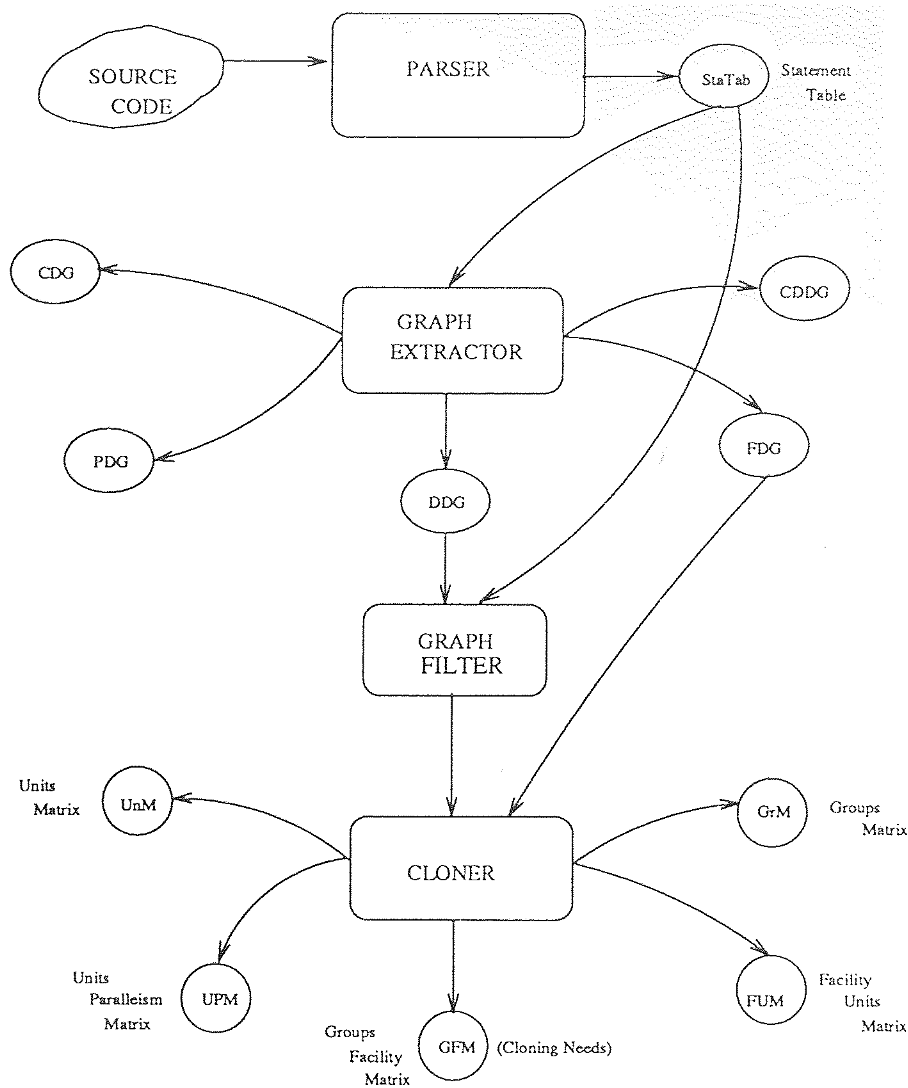


Figure 5.2 System Design of The Compiler at *level 2*

and it contains the cloning requirements of the different facilities in the application. The matrix is generated as the *module_name.clone* file. The complete design of the compiler is shown in Figure 5.2.

CHAPTER 6

CONTRIBUTIONS TO KNOWLEDGE

The potential of ADT modules for reusability is made ineffective to a large extent by their inefficiencies at run time. The ARPC model of parallel execution, when applied to programs constructed out of ADT modules in conjunction with the cloning techniques, can significantly enhance the run time performance of such programs. Extending the dependence graphs of programs to include code dependence is found to reveal greater opportunities for concurrent execution. Implementations of the algorithms for graph extraction at compile-time proved these facts. Further more, by subjecting the graphs to cloning analysis at most by link-time, an upper bound on the number of clones that could be used could be determined. Algorithms for handling conditional statements (through transformations) and loops (by unrolling) were designed and implemented and was found to enhance the accuracy of the clone analysis. These are the main contributions of this work. The parallelism information so extracted, could be used for constructing a feasible schedule statically, and this could be of importance to hard real-time systems where timing constraints are a concern.

APPENDIX A

GRAMMAR OF RT-RESOLVE

Realization Module

```
START:
        PROCESS |
        MODULE |
        CONTROL_PROCESS
PROCESS:
        PROCESS_TOKEN PROCESS_NAME
          OPT_PROCESS_PARM_SEC
          OPT_VAR_DECL_SEC
          OPT_FAC_DECL_SEC
          PROCESS_PROC_DECL
        END_TOKEN PROCESS_NAME
OPT_PROCESS_PARM_SEC:
        | PARM_TOKEN OPT_DEADLINE OPT_FRAME
        | PARM_TOKEN OPT_FRAME OPT_DEADLINE
OPT_DEADLINE:
        |
        DEADLINE_TOKEN DEADLINE
OPT_FRAME:
        |
        FRAME_TOKEN FRAME
CONTROL_PROCESS:
        CNTRL_TOKEN PROCESS_TOKEN PROCESS_NAME
```

```
        BEGIN_TOKEN
          PROCESS_DECL_SEQ
        END_TOKEN PROCESS_NAME
PROCESS_DECL_SEQ:
        PROCESS_DECL
        | PROCESS_DECL_SEQ PROCESS_DECL
PROCESS_DECL:
        PROCESS_NAME OPT_PROCESS_ARGS SEMICOLN_TOKEN
OPT_PROCESS_ARGS:
        |
        LPREN_TOKEN DEADLINE COMA_TOKEN FRAME
        RPREN_TOKEN
DEADLINE:
        INT_TOKEN
FRAME:
        INT_TOKEN
MODULE:
        MOD_TOKEN MOD_NAME
          NUM_OPS
          OPT_MOD_PARM_SEC
          OPT_AUX_SEC
          OPT_INTF_SEC
        END_TOKEN MOD_NAME
```

Module Parameter Section

```
NUM_OPS:
```

```
        NUM_TOKEN OPR_TOKEN ASS_TOKEN INT_TOKEN
        SEMICOLN_TOKEN
OPT_MOD_PARM_SEC:
        |
        MOD_PARM_SEC
MOD_PARM_SEC:
        MOD_TOKEN PARM_TOKEN
          MOD_PARM_SEQ
        END_TOKEN MOD_TOKEN PARM_TOKEN
MOD_PARM_SEQ:
        MOD_PARM SEMICOLN_TOKEN
        | MOD_PARM_SEQ MOD_PARM SEMICOLN_TOKEN
MOD_PARM:
        TYPE_TOKEN TYPE_NAME
        | PROC_HDR
        | FUNC_HDR
        | CTRL_HDR
PROC_HDR:
        PROC_TOKEN PROC_NAME
        OPT_PROC_PARM_SEC
        END_TOKEN PROC_NAME
FUNC_HDR:
        FUNC_TOKEN FUNC_NAME RETS_TOKEN VAR_NAME
        COLN_TOKEN
          LONG_TYPE_NAME
          OPT_FUNC_CTRL_PARM_SEC
        END_TOKEN FUNC_NAME
```

CTRL_HDR:

CNTRL_TOKEN CTRL_NAME

OPT_FUNC_CTRL_PARM_SEC

END_TOKEN CTRL_NAME

OPT_PRVD_TYPES:

|

PRVD_TYPES_SEC

PRVD_TYPES_SEC:

PRVD_TOKEN TYPS_TOKEN

PRVD_TYPES_SEQ

END_TOKEN

PRVD_TYPES_SEQ:

PRVD_TYPE

|

PRVD_TYPES_SEQ PRVD_TYPE

PRVD_TYPE:

TYPE_NAME IS_TOKEN REPRESENT_TOKEN BY_TOKEN

LONG_TYPE_NAME SEMICOLN_TOKEN

Auxiliary Section

OPT_AUX_SEC:

|

AUX_SEC

AUX_SEC:

AUX_TOKEN

OPT_FAC_DECL_SEC

    OPT_PRVD_TYPES
    OPT_VAR_DECL_SEC
    OPT_AUX_OPR_DECL_SEC
    OPT_REAL_AUX_INIT
END_TOKEN AUX_TOKEN

Facility Declaration Section

OPT_FAC_DECL_SEC:

|

FAC_DECL_SEC

FAC_DECL_SEC:

FAC_TOKEN

FAC_DECL_SEQ

END_TOKEN FAC_TOKEN

FAC_DECL_SEQ:

FAC_DECL

| FAC_DECL_SEQ

FAC_DECL

FAC_DECL:

FAC_NAME IS_TOKEN MOD_NAME OPT_FAC_ARG_LIST

SEMICOLN_TOKEN

OPT_FAC_ARG_LIST:

|

LPREN_TOKEN FAC_ARG_LIST RPREN_TOKEN

FAC_ARG_LIST:

FAC_ARG

| FAC_ARG_LIST COMA_TOKEN FAC_ARG

FAC_ARG:

LONG_NAME

Auxiliary Operation Declaration Section

OPT_AUX_OPR_DECL_SEC:

|

AUX_OPR_DECL_SEC

AUX_OPR_DECL_SEC:

OPR_TOKEN

OPR_DECL_SEQ

END_TOKEN OPR_TOKEN

OPT_REAL_AUX_INIT:

|

REAL_AUX_INIT

REAL_AUX_INIT:

INIT_TOKEN

BEGIN_TOKEN

OPT_LOC_VAR_DECL_SEC

CODE

END_TOKEN INIT_TOKEN

Interface Section

OPT_INTF_SEC:

|

INTF_SEC

INTF_SEC:

INTF_TOKEN

OPT_TYPE_DECL_SEQ

OPT_OPR_DECL_SEQ

END_TOKEN INTF_TOKEN

Provided Types

OPT_TYPE_DECL_SEQ:

|

TYPE_DECL_SEQ

TYPE_DECL_SEQ:

TYPE_DECL

| TYPE_DECL_SEQ

TYPE_DECL

TYPE_DECL:

TYPE_TOKEN TYPE_NAME IS_TOKEN REPRESENT_TOKEN

BY_TOKEN

LONG_TYPE_NAME EXEMPLAR_TOKEN VAR_NAME

OPT_TYPE_INIT

OPT_TYPE_FIN

END_TOKEN TYPE_NAME

OPT_TYPE_INIT:

|

TYPE_INIT

TYPE_INIT:

```
        INIT_TOKEN
        BEGIN_TOKEN
          OPT_LOC_VAR_DECL_SEC
          CODE
        END_TOKEN INIT_TOKEN
OPT_TYPE_FIN:
        |
        TYPE_FIN
TYPE_FIN:
        FIN_TOKEN
        BEGIN_TOKEN
          OPT_LOC_VAR_DECL_SEC
          CODE
        END_TOKEN FIN_TOKEN
OPT_LOC_VAR_DECL_SEC:
        |
        LOC_VAR_DECL_SEC
OPT_VAR_DECL_SEC :
        |
        VAR_DECL_SEC
VAR_DECL_SEC:
        VAR_TOKEN
          VAR_DECL_SEQ
        END_TOKEN VAR_TOKEN
VAR_DECL_SEQ:
        VAR_DECL
        | VAR_DECL_SEQ VAR_DECL
```

VAR_DECL:

VAR_NAME COLN_TOKEN LONG_TYPE_NAME

SEMICOLN_TOKEN

LOC_VAR_DECL_SEC:

LOCAL_TOKEN VAR_TOKEN

LOC_VAR_DECL_SEQ

END_TOKEN LOCAL_TOKEN VAR_TOKEN

LOC_VAR_DECL_SEQ:

LOC_VAR_DECL

| LOC_VAR_DECL_SEQ LOC_VAR_DECL

LOC_VAR_DECL:

VAR_NAME COLN_TOKEN LONG_TYPE_NAME

SEMICOLN_TOKEN

OPT_OPR_DECL_SEQ:

|

OPR_DECL_SEQ

OPR_DECL_SEQ:

PROC_DECL |

FUNC_DECL |

CTRL_DECL |

OPR_DECL_SEQ

PROC_DECL |

OPR_DECL_SEQ

FUNC_DECL|

OPR_DECL_SEQ

CTRL_DECL

PROC_DECL:

```
        PROC_TOKEN PROC_NAME
        OPT_PROC_PARM_SEC
        BEGIN_TOKEN
          OPT_LOC_VAR_DECL_SEC
          CODE
        END_TOKEN PROC_NAME
PROCESS_PROC_DECL:
        PROC_TOKEN PROC_NAME
          OPT_PROC_PARM_SEC
          BEGIN_TOKEN
            OPT_LOC_VAR_DECL_SEC
            CODE
          END_TOKEN PROC_NAME
FUNC_DECL:
        FUNC_TOKEN FUNC_NAME RETS_TOKEN VAR_NAME
        COLN_TOKEN LONG_TYPE_NAME
          OPT_FUNC_CTRL_PARM_SEC
          BEGIN_TOKEN
            OPT_LOC_VAR_DECL_SEC
            CODE
          END_TOKEN FUNC_NAME
CTRL_DECL:
        CNTRL_TOKEN CTRL_NAME
          OPT_FUNC_CTRL_PARM_SEC
          BEGIN_TOKEN
            OPT_LOC_VAR_DECL_SEC
            CODE
```

END_TOKEN CTRL_NAME

OPT_PROC_PARM_SEC:

|

PROC_PARM_SEC

PROC_PARM_SEC:

PARM_TOKEN

PROC_PARM_SEQ

END_TOKEN PARM_TOKEN

PROC_PARM_SEQ:

PROC_PARM

| PROC_PARM_SEQ PROC_PARM

PROC_PARM:

PRESV_TOKEN VAR_NAME COLN_TOKEN LONG_TYPE_NAME

SEMICOLN_TOKEN

|

ALT_TOKEN VAR_NAME COLN_TOKEN LONG_TYPE_NAME

SEMICOLN_TOKEN

|

PROD_TOKEN VAR_NAME COLN_TOKEN LONG_TYPE_NAME

SEMICOLN_TOKEN

|

CONSU_TOKEN VAR_NAME COLN_TOKEN LONG_TYPE_NAME

SEMICOLN_TOKEN

OPT_FUNC_CTRL_PARM_SEC:

|

FUNC_CTRL_PARM_SEC

FUNC_CTRL_PARM_SEC:

PARM_TOKEN

FUNC_CTRL_PARM_SEC

END_TOKEN PARM_TOKEN

FUNC_CTRL_PARM_SEQ:

FUNC_CTRL_PARM

| FUNC_CTRL_PARM_SEQ FUNC_CTRL_PARM

FUNC_CTRL_PARM:

PRESV_TOKEN VAR_NAME COLN_TOKEN LONG_TYPE_NAME

SEMICOLN_TOKEN

CODE:

STMT SEMICOLN_TOKEN |

CODE STMT SEMICOLN_TOKEN

STMT:

SWAP

| ASSIGN

| IF

| WHILE

| RETURN

| DO

| PROC_CALL

SWAP:

VAR_NAME COLN_TOKEN ASS_TOKEN COLN_TOKEN

VAR_NAME

ASSIGN:

VAR_NAME COLN_TOKEN ASS_TOKEN FUNC_CALL

IF:

IF_TOKEN OPT_NOT CTRL_CAL THEN_TOKEN

```
      CODE
      OPT_ELSE
    END_TOKEN IF_TOKEN
OPT_NOT:
    NOT_TOKEN
    |
OPT_ELSE:
    |
    ELSE_TOKEN
    CODE
WHILE:
    WHILE_TOKEN NOT_TOKEN CTRL_CALL DO_TOKEN
      CODE
    END_TOKEN WHILE_TOKEN
    |
    WHILE_TOKEN CTRL_CALL DO_TOKEN
      CODE
    END_TOKEN WHILE_TOKEN
DO:
    DO_TOKEN INT_TOKEN TIMES_TOKEN
      BEGIN_TOKEN
      CODE
    END_TOKEN DO_TOKEN
RETURN:
    RET_TOKEN
    | RET_TOKEN TRUE_TOKEN
    | RET_TOKEN FALSE_TOKEN
```

PROC_CALL:

LONG_PROC_NAME LPREN_TOKEN OPT_ARG_LIST

RPREN_TOKEN

| LONG_PROC_NAME

FUNC_CALL:

LONG_FUNC_NAME LPREN_TOKEN OPT_ARG_LIST

RPREN_TOKEN

| LONG_FUNC_NAME

CTRL_CALL:

LONG_CTRL_NAME LPREN_TOKEN OPT_ARG_LIST

RPREN_TOKEN

| LONG_CTRL_NAME

OPT_ARG_LIST:

|

ARG_LIST

ARG_LIST:

VAR_NAME

| ARG_LIST COMA_TOKEN VAR_NAME

LONG_TYPE_NAME:

LONG_NAME1

LONG_PROC_NAME:

LONG_NAME

LONG_FUNC_NAME:

LONG_NAME

LONG_CTRL_NAME:

LONG_NAME

LONG_NAME1:

NAME

LONG_NAME1:

FAC_NAME DOT_TOKEN NAME

LONG_NAME:

NAME

LONG_NAME:

FAC_NAME DOT_TOKEN NAME

NAME:

ID_TOKEN

TYPE_NAME:

ID_TOKEN

PROC_NAME:

ID_TOKEN

FUNC_NAME:

ID_TOKEN

CTRL_NAME:

ID_TOKEN

VAR_NAME:

ID_TOKEN

MOD_NAME:

ID_TOKEN

FAC_NAME:

ID_TOKEN

PROCESS_NAME:

ID_TOKEN

APPENDIX B

PRIMITIVE MODULE OPERATIONS

Integer Operations

1. procedure increment(alters i:int)

   ENSURES: $i = \#i + 1$

2. function add(preserves i: int; preserves j: int) returns x: int

   ENSURES: $x = i + j$

3. function subtract(preserves i: int; preserves j: int) returns x: int

   ENSURES: $x = i - j$

4. function multiply(preserves i:int; preserves j: int) returns x: int

   ENSURES: $x = i * j$

5. function divide(preserves i:int; preserves j:int) returns x: int

   ENSURES: $x = i / j$

6. control less_than_or_equal(preserves i: int; preserves j: int)

   ENSURES: less_than_or_equal iff $i \leq j$

7. control equal(preserves i: int; preserves j: int)

   ENSURES: equal iff $i = j$

8. function get_min_int returns x: int

   ENSURES: x *= the minimum integer value allowed*

9. function get_max_int returns x: int

   ENSURES: x *= the maximum integer value allowed*

10. function integer_copy (preserves i:int) returns j:int

    ENSURES: j=i

11. function gen_one returns one: int

    ENSURES: one=1

12. function gen_five returns five: int

    ENSURES: five=5

13. function read returns x: int

    ENSURES: x = next value in input stream

14. procedure write(preserves x: int)

    ENSURES: x is appended to output stream

15. function integer_initialize returns i: int

    ENSURES: i=0

16. procedure integer_finalize(alters i: int)

    ENSURES: storage is reclaimed for i

Array Operations

1. procedure access(alters a: array, preserves position: int, alters item: int)

   ENSURES: a(position)=#item and item=#a(position)

2. procedure set_max_size(alters a:array, preserves size: int)

   ENSURES: a.size=size and a(i)=INIT(int), for $0<i<size+1$

3. function get_max_size(preserves a:array) returns size: int

   ENSURES: size=a.size

4. procedure array_initialize returns a: array

   ENSURES: a.size=0

5. procedure array_finalize(alters a: array)

   ENSURES: storage is reclaimed for a, and each element of a is finalized

APPENDIX C

VEHICLE APPLICATION PROGRAM

```
control process main
  begin
    vehicledr(3000,3000);
  end main


process vehicledr
    facilities
      i is integer;
      v is vehicle;
    end facilities
    procedure vehicledr
      begin
        local variables
          veh1 : v.vehicleType;
          veh2 : v.vehicleType;
          id1 : i.integer;
          id2 : i.integer;
          rate : i.integer;
          vh1x : i.integer;
          vh1y : i.integer;
          vh2x : i.integer;
          vh2y : i.integer;
          item : i.integer;
```

```
   one : i.integer;
   two : i.integer;
   three: i.integer;
   four : i.integer;
   five : i.integer;
   six : i.integer;
   time : i.integer;
   X : i.integer;
   Y : i.integer;
end local variables
   two := i.gen_one;
   i.increment(two);
   one := i.gen_one;
   three := i.add_opr(one,two);
   four := i.gen_five;
   i.decrement(four);
   five := i.gen_five;
   six := i.gen_five;
   i.increment(six);
   rate := i.gen_one;
   i.increment(rate);
   id1 := i.gen_one;
   vh1x := i.gen_five;
   i.decrement(vh1x);
   i.decrement(vh1x);
   vh1y := i.multiply(two,four);
   id2 := i.gen_one;
```

```
            i.increment(id2);
            vh2x := i.multiply(two,six);
            vh2y := i.multiply(three,five);
            v.InputVehicle(veh1,id1,vh1x,vh1y);
            v.AccessVId(veh1,item);
            v.AccessVXY(veh1,X,Y);
            i.integer_write(item);
            i.integer_write(X);
            i.integer_write(Y);
            v.InputVehicle(veh2,id2,vh2x,vh2y);
            v.AccessVId(veh2,item);
            v.AccessVXY(veh2,X,Y);
            i.integer_write(item);
            i.integer_write(X);
            i.integer_write(Y);
            v.FindDistance(veh1,veh2,rate,time);
            i.integer_write(time);
        end vehicledr

module coordinate
  num operations = 6;
  auxiliary
    facilities
      i is integer;
      in is integer;
      re is record2(i.integer, in.integer);
    end facilities
```

```
provided types
  coordtype is represented by re.record2;
end
end auxiliary
interface
type coordtype is represented by re.record2 exemplar ex
end coordtype
procedure accessX
  parameters
    preserves RXY: coordtype;
    produces X: i.integer;
  end parameters
  begin
    local variables
      Z : i.integer;
    end local variables
      re.rec1_access(RXY, Z);
      X := i.integer_copy(Z);
      re.rec1_access(RXY, Z);
  end accessX
procedure accessY
  parameters
    preserves RXY: coordtype;
    produces Y: in.integer;
  end parameters
  begin
    local variables
```

```
            Z : in.integer;
        end local variables
            re.rec2_access(RXY, Z);
            Y := in.integer_copy(Z);
            re.rec2_access(RXY,Z);
    end accessY
procedure inputCoord
    parameters
        alters Coord: coordtype;
        preserves X : i.integer;
        preserves Y : in.integer;
    end parameters
    begin
            re.rec1_access(Coord,X);
            re.rec2_access(Coord,Y);
    end inputCoord
procedure getDistance
    parameters
        preserves C1: coordtype;
        preserves C2: coordtype;
        produces dist: in.integer;
    end parameters
    begin
        local variables
            X1: in.integer;
            Y1: in.integer;
            X2: in.integer;
```

```
        Y2: in.integer;
        DX: in.integer;
        DY: in.integer;
      end local variables
        accessX(C1,X1);
        accessY(C1,Y1);
        accessX(C2,X2);
        accessY(C2,Y2);
        DX := in.subtract(X2,X1);
        DY := in.subtract(Y2,Y1);
        dist := in.add_opr(DX,DY);
    end getDistance
procedure travelTime
    parameters
      preserves C1: coordtype;
      preserves C2: coordtype;
      preserves Rate: in.integer;
      produces time: in.integer;
    end parameters
    begin
      local variables
        dist: in.integer;
      end local variables
        getDistance(C1,C2,dist);
        time:= in.divide(dist,Rate);
    end travelTime
control coordsEqual
```

```
parameters
  preserves C1: coordtype;
  preserves C2: coordtype;
  preserves C2: coordtype;
end parameters
begin
  local variables
    X1: i.integer;
    Y1: in.integer;
    X2: i.integer;
    Y2: in.integer;
  end local variables
    accessX(C1,X1);
    i.increment(X1);
    accessY(C1,Y2);
    Y2 := in.integer_copy(Y1);
    if i.equal(X1,X2) then
      accessX(C2,X2);
      if in.equal(Y1,Y2) then
        re.rec1_access(C1,X2);
        re.rec2_access(C2,Y1);
      else
        re.rec1_access(C2,X1);
        re.rec2_access(C1,Y2);
      end if;
      Y1 := in.gen_five;
      return true;
```

```
            else
               re.rec1_access(C1,X1);
               return false;
            end if;
        end coordsEqual
    end interface
end coordinate


module vehicle
    num operations = 5;
    auxiliary
        facilities
            in is integer;
            co is coordinate;
            re is record2(in.integer, co.coordtype);
        end facilities
        provided types
            vehicleType is represented by re.record2;
        end
    end auxiliary
    interface
        type vehicleType is represented by re.record2 exemplar ex
        end vehicleType
        procedure AccessVId
            parameters
                preserves vtype: vehicleType;
                produces id: in.integer;
```

```
    end parameters
    begin
      local variables
        temp : in.integer;
      end local variables
        re.rec1_access(vtype,temp);
        id := in.integer_copy(temp);
        re.rec1_access(vtype,temp);
    end AccessVId
  procedure AccessVXY
    parameters
      preserves vtype: vehicleType;
      produces X: in.integer;
      produces Y: in.integer;
    end parameters
    begin
      local variables
        temp : co.coordtype;
      end local variables
        re.rec2_access(vtype,temp);
        co.accessX(temp,X);
        co.accessY(temp,Y);
        re.rec2_access(vtype,temp);
    end AccessVXY
  procedure AccessVXYC
    parameters
      preserves vtype: vehicleType;
```

```
    produces C: co.coordtype;
  end parameters
  begin
    local variables
      temp : co.coordtype;
      X : in.integer;
      Y : in.integer;
    end local variables
      re.rec2_access(vtype,temp);
      co.accessX(temp,X);
      co.accessY(temp,Y);
      co.inputCoord(C,X,Y);
      re.rec2_access(vtype,temp);
  end AccessVXY
procedure InputVehicle
  parameters
    alters vehicle: vehicleType;
    preserves id : in.integer;
    preserves x : in.integer;
    preserves y : in.integer;
  end parameters
  begin
    local variables
      cord : co.coordtype;
      item : in.integer;
    end local variables
      co.inputCoord(cord,x,y);
```

```
        re.rec1_access(vehicle,id);
        re.rec2_access(vehicle,cord);
      end InputVehicle
    procedure FindDistance
      parameters
        preserves veh1: vehicleType;
        preserves veh2: vehicleType;
        preserves rate: in.integer;
        produces time : in.integer;
      end parameters
      begin
        local variables
          c1 : co.coordtype;
          c2 : co.coordtype;
          dist : in.integer;
        end local variables
          AccessVXYC(veh1,c1);
          AccessVXYC(veh2,c2);
          co.getDistance(c1,c2,dist);
          in.integer_write(dist);
          co.travelTime(c1,c2,rate,time);
    end FindDistance
  end interface
end vehicle


module pcomp
  num operations = 5;
```

```
auxiliary
  facilities
    in is integer;
    op is op;
    q is queue(op.optype);
    re is record2(in.integer, q.queue);
  end facilities
provided types
  pcompType is represented by re.record2;
  seqType is represented by q.queue;
end
end auxiliary
interface
  type pcompType is represented by re.record2 exemplar ex
  end pcompType
  type seqType is represented by q.queue exemplar eq
  end seqType
  procedure AccessCompID
    parameters
      preserves comp: pcompType;
      produces id: in.integer;
    end parameters
    begin
      local variables
        temp : in.integer;
      end local variables
        re.rec1_access(comp, temp);
```

```
      id := in.integer_copy(temp);
      re.recl_access(comp, temp);
  end AccessCompID
procedure AccessSeq
  parameters
    preserves comp: pcompType;
    produces seq: seqType;
  end parameters
  begin
    local variables
      temp : op.optype;
      diff : in.integer;
      newq : q.queue;
    end local variables
      diff := in.gen_five;
      q.SetSize(seq,diff);
      while not q.IsEmpty(newq) do
        q.dequeue(newq,temp);
        q.enqueue(seq,temp);
      end while;
  end AccessSeq
procedure InputSeq
  parameters
    alters Seq: seqType;
    consumes x : op.optype;
  end parameters
  begin
```

```
        q.enqueue(Seq,x);
      end InputSeq
    procedure DeleteSeq
      parameters
        alters Seq: seqType;
        produces x : op.optype;
      end parameters
      begin
        q.dequeue(Seq,x);
      end DeleteSeq
    control SeqEmpty
      parameters
        preserves seq: seqType;
      end parameters
      begin
      if q.IsEmpty(seq)
        then return true;
      else return false;
      end if;
      end SeqEmpty
  end interface
end pcomp
```

APPENDIX D

CONDITIONAL TRANSFORMATIONS

The data dependence graphs (*DDGs*) and the different matrices generated i.e., *Units*, *Groups*, *Units Parallelism Matrix (UPM)*, *Facility Units Matrix (FUM)* & *Group Facility Matrix (GFM)* during the transformation of conditionals at two levels of the application program (*coordsEqual operation, coordinate module*) discussed in Chapter 4, is illustrated in this Appendix.

We first show the subgraphs which are being subjected to the cloning analysis, before illustrating the generated matrices. Extraction of the subgraphs discussed in this Appendix have already been detailed in Chapter 4. Note that, the algorithms for transformation are applied at the two different levels of the conditional and after each transformation, the conditional body is replaced with a single supernode. The convention for representing the statements in the graphs i.e., as *Sn*, where *n* is the statement label number, is also adopted here. Further, a *Unit* is denoted as *Unx*, where *x* is the unit number, a *Group* as *Grx*, where *x* is the group number. The *facilities* are simply represented by their names. Facilities used by the statements in the subgraphs are *i*, *in* and *re*.

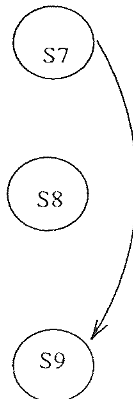


Figure D.1 Left Graph of Conditional at level 2

Table D.1 Units from Left Graph (Conditional at level 2)

-	-	-
Un1	S7	S9
Un2	S8	

Table D.2 Units Parallelism Matrix of Left Graph (level 2)

-	Un1	Un2
Un1	-	1
Un2		-

Table D.3 Groups of Units of Left Graph (level 2)

-	-	-
Gr1	Un1	Un2
Gr2	Un2	

Table D.4 Facility Units Matrix of Left Graph (level 2)

-	Un1	Un2
i		
in	1	
re	1	1

Table D.5 Group Facility Matrix of Left Graph (level 2)

-	i	in	re
Gr1	0	1	2
Gr2	0	0	1
Max	0	1	2

Table D.6 Units from Right Graph (Conditional at level 2)

-	-	-
Un1	S7	S11
Un2	S10	

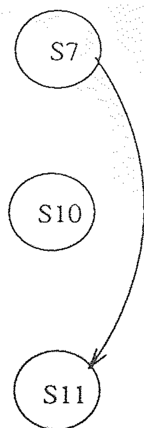


Figure D.2 Right Graph of Conditional at level 2

Table D.7 Units Parallelism Matrix of Right Graph (level 2)

_	Un1	Un2
Un1	_	1
Un2		_

Table D.8 Groups of Units of Right Graph (level 2)

_	_	_
Gr1	Un1	Un2
Gr2	Un2	

Table D.9 Facility Units Matrix of Right Graph (level 2)

_	Un1	Un2
i		
in	1	
re	1	1

Table D.10 Group Facility Matrix of Right Graph (level 2)

_	i	in	re
Gr1	0	1	2
Gr2	0	0	1
Max	0	1	2

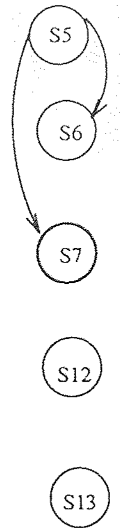


Figure D.3 Left Graph of Conditional at level 1

Table D.11 Units from Left Graph (Conditional at level 1)

_	_
Un1	S5
Un2	S6
Un3	S7
Un4	S12
Un5	S13

Table D.12 Units Parallelism Matrix of Left Graph (level 1)

_	Un1	Un2	Un3	Un4	Un5
Un1	_	0	0	1	1
Un2		_	1	1	1
Un3			_	1	1
Un4				_	1
Un5					_

Table D.13 Groups of Units of Left Graph (level 1)

_	_	_	_	_
Gr1	Un1	Un4	Un5	
Gr2	Un2	Un3	Un4	Un5
Gr3	Un4	Un5		
Gr4	Un5			

Table D.14 Facility Units Matrix of Left Graph (level 1)

_	Un1	Un2	Un3	Un4	Un5
i	1		0		
in			1	1	
re			2		

Table D.15 Group Facility Matrix of Left Graph (level 1)

_	i	in	re
Gr1	1	1	0
Gr2	0	2	2
Gr2	0	1	0
Gr2	0	0	0
Max	1	2	2

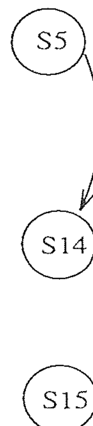


Figure D.4 Right Graph of Conditional at level 1

Table D.16 Units from Right Graph (Conditional at level 1)

_	_	_
Un1	S5	S14
Un2	S15	

Table D.17 Units Parallelism Matrix of Right Graph (level 1)

_	Un1	Un2
Un1	_	1
Un2		_

Table D.18 Groups of Units of Right Graph (level 1)

_	_	_
Gr1	Un1	Un2
Gr2	Un2	

Table D.19 Facility Units Matrix of Right Graph (level 1)

_	Un1	Un2
i	1	
in		
re	1	

Table D.20 Group Facility Matrix of Right Graph (level 1)

_	i	in	re
Gr1	1	0	1
Gr2	0	0	0
Max	1	0	1

REFERENCES

1. L. R. Welch. "Cloning ADT modules to increase parallelism: Rationale and techniques," In *Fifth IEEE Symposium on Parallel and Distributed Computing*. IEEE, December 1993.

2. K. D. Cooper, M. W. Hall, and K. Kennedy. "A methodology for procedure cloning," *Computer Languages*, 19(2):105–117, April 1993.

3. L. R. Welch. "Assignment of ADT modules to processors," In *The International Parallel Processing Symposium*. IEEE, March 1992.

4. D. Harms and B.W. Weide. "Types, copying, and swapping: Their influences on the design of reusable software components," *IEEE Transactions on Software Engineering*, 17(5):424–435, May 1991.

5. L. R. Welch. "Architectural Support for, and Parallel Execution of, Programs Constructed from Reusable Software Components," *PhD Thesis*, The Ohio State University, December 1990.

6. 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.

7. R. A. Steigerwald, C. A. Warack and D. A. Cook. "Issues in Integrating Reusable Ada 9X Objects into Distributed Real-Time Systems." In *The Second Workshop on Parallel and Distributed Real-Time Systems*. IEEE, April 28-29, Cancun, Mexico 1994.

8. 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.

9. E. C. Cooper. "Circus: A replicated procedure call facility." In *Proc. 4th Symp. Reliability in Distributed Software and Databases*, pages 11–24, 1984.

10. F. Bastani, W. Hilal, and S. S. Iyengar. "Efficient abstract data type components for distributed and parallel systems," *Computer*, 20(10):33-44, October 1987.

11. L. R. Welch, A. D. Stoyenko, and S. Chen. "Assignment of ADT modules with random neural networks," In *Proceedings of Hawaii International Conference on System Sciences*, pages 546–555. IEEE, January 1993.

12. D. Padua and M. Wolfe. "Advanced compiler optimizations for supercomputers," *CACM*, 29(12):1184-1201, December 1986.

13. Guohui Yu. "Use of concurrency enhancement in off-line schedule construction," In *The Second Workshop on Parallel and Distributed Real-Time Systems*, IEEE, April 28-29, Cancun, Mexico 1994.

14. W. Rossak, A. D. Stoyenko, and L. R. Welch. "THE COMPONENT MANAGER - A Hybrid Reuse-Tool Supporting Interactive and Automated Retrieval of Software Components," Technical Report CIS-92-04, New Jersey Institute of Technology, 1992.

15. M. Sitaraman, L. R. Welch, and D. E. Harms. "On Specification of Reusable Software Components," In *The International Journal of Software Engineering and Knowledge Engineering*, 3(2), 1993.

16. R. A. Steigerwald and L. R. Welch. "Reusable Component Retrieval for Real-Time Applications," In *Proceedings of The First IEEE Workshop on Real-Time Applications*, May, 1993.

17. 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," Technical Report CIS-93-16, New Jersey Institute of Technology, December, 1993

18. J. P. C. Verhoosel, L. R. Welch, D. K. Hammer and A. D. Stoyenko. "A Model for Scheduling of Object-Based, Hard Real-Time Parallel Processes," *Journal of Real-Time Systems*, 1994 (to appear).

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

20. L. R. Welch. "A Parallel Virtual Machine for Programs Composed of Abstract Data Types," *IEEE Transactions on Computers*, accepted for publication-to appear.

21. L. R. Welch, A. L. Samuel, M. W. Masters, R. D. Harrison, A. D. Stoyenko, and J. Caruso. "A Framework for Automated Reengineering of Complex Computer Systems," In *Proceedings of The Fourth Systems Reengineering Technology Workshop*, pages 44-56, Naval Surface Warfare Center, February 1994.

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