DEPENDENCE ANALYSIS FOR INFERRING INFORMATION FLOW PROPERTIES IN SPARK ADA PROGRAMS

by

HARIHARAN THIAGARAJAN

B.Tech., Anna University, 2009

A THESIS

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas
2011

Approved by:

Major Professor
John Hatcliff

Copyright

Abstract

With the increase in development of safety and security critical systems, it is important to have more sophisticated methods for engineering such systems. It can be difficult to understand and verify critical properties of these systems because of their ever growing size and complexity. Even a small error in a complex system may result in casualty or significant monetary loss. Consequently, there is a rise in the demand for scalable and accurate techniques to enable faster development and verification of high assurance systems.

This thesis focuses on discovering dependencies between various parts of a system and leveraging that knowledge to infer information flow properties and to verify security policies specified for the system. The primary contribution of this thesis is a technique to build dependence graphs for languages which feature abstraction and refinement. Inter-procedural slicing and inter-procedural chopping are the techniques used to analyze the properties of the system statically.

The approach outlined in this thesis provides a domain-specific language to query the information flow properties and to specify security policies for a critical system. The specified policies can then be verified using existing static analysis techniques. All the above contributions are integrated with a development environment used to develop the critical system. The resulting software development tool helps programmers develop, infer, and verify safety and security systems in a single unified environment.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

Computers today have become an integral part of our life. We have started relying on computers for security and safety critical systems such as medical care, various kinds of transportation and national security. It is challenging to engineer such systems, since failure can mean huge losses and the magnitude of such loss sets them apart from the less critical systems.

Examples of *safety critical* programs are avionics programs, systems employed in automobile brakes, railway signals, medical equipment, etc. Failure in any of these systems may lead to loss of life or a considerable amount of property damage.

Examples of *security critical* programs are systems that handle communication and data management of highly confidential data, military support systems, etc. Failure in such systems could lead to a compromise in national security or a bank theft.

Both safety and security critical systems are designed with great care along with specialized hardware support. The integrity of the entire system relies on the correctness of the software. Programs are said to be correct when it matches the expected behavior defined in its requirement documentation. In other words, a program functions exactly according to the specification, nothing more, nothing less. Both safety critical and security critical systems can be collectively addressed as high assurance systems.

High assurance systems are those that require convincing evidence that the system adequately addresses critical properties such as security and safety[5]. These type of systems requires a high level of rigor in both analysis and design. Multiple Independent Levels of Security and Safety (MILS) is an approach used to design, construct, integrat, and evaluate high assurance embedded systems. There are two steps in MILS approach: first, a security policy architecture is developed in which the interacting components are separated in such a manner so that the trusted components are as simple as possible; second, the components of the policy architecture are allocated to securely shared resources[13].

Rockwell Collins, a pioneer company in the field of aerospace and defense, has developed a high assurance solution called SecureOne Cross Domain Technologies. It is a family of high assurance technologies for military tactical systems, which enable increased situational awareness through trusted multi-classification information sharing. It is designed and built according to the MILS architecture[1].

1.2 Problem

Our goal is to implement a security policy verification tool. The precise interpretation of security for a given system is called its 'security policy'[13]. Defining and verifying the system against its security policy is a challenging task in a MILS approach.

1.2.1 Information Flow

In order to perform security analysis the first step is to analyze the information flow of the system. Analyzing individual functionality is much simpler than analyzing the entire system. 'spark' is a programing language used to develop high assurance systems. Computing the information flow manually is tedious and error prone. Inferring of information flow is difficult when it is not presented in a concise manner.

1.2.2 Visualization of Information flow

A developer needs a thorough understanding of the information flow in order to write better security policies. one of the best ways to present information flow to the developer is a visual representation. To implement this requirement various factors like scalability, precision of the result and robustness of the tool are to be considered.

1.2.3 Policy Language

There are very few existing policy languages to specify the security policies for an embedded system. A domain specific language is required to efficiently specify security policies. Developing a new language in a relatively unexplored domain requires a considerable amount of research.

1.2.4 Integrated Development Environment

Incorporating the security policy verification tool in an Integrated Development Environment (IDE) requires considerable effort to customize the tools according the IDE requirements. An IDE helps in visualizing the information flow and also to indicate the program points on which the security policy fails. Development, verification and specifying policies in a single environment saves considerable amount of engineering time.

1.3 Thesis

This thesis provides a technique to infer and check the information flow properties of Spark programs. This thesis is organized to complete these goals in the order listed.

- Introduce the spark ada language and its applications
- Information flow analysis for Spark language
- Architecture of the policy language verification tool

- System dependent graph for a spark language program
- Slicing techniques with example
- Chopping techniques with example
- Formal representation of policy language syntax
- Verification technique of the security policy

1.4 Solution

This thesis presents a number of solutions:

- The system wide information flow analysis in Spark language
- A domain specific language to specify security policy
- Verifying the policy against the system
- Integrations of the above in a development environment

This thesis addresses the information flow annotations of the Spark language. The spark examiner's information flow analyses are procedural-level analysis and it only captures flow from return values to input values in a procedure. However, the information flow technique presented in this thesis is capable of performing system-level analysis. It can also compute information flow from any statement from the entire system.

This thesis presents a policy language to specify the security policies of the system.Unlike existing policy languages which have targeted a specific programming language, this policy language is completely independent of spark.An Eclipse IDE has been developed to help in engineering the system according to the specified security policies, and to verify the correctness.

Many of the contributions originated from Dr. John Hatcliff, Dr. Robby and Dr. Torben Amtoft. Dr. Hatcliff and Dr. Robby provided the initial information flow analysis framework based on which the above-mentioned tools are built.

Chapter 2

Literature Review

This chapter presents all the background and relevant works to this thesis. As the tool described in this thesis is for Spark language. the spark language features are discussed under section 2.1. The dependence analyses described in this thesis is built on various other analyses, and the existing intra-procedural analysis is improved to inter-procedural analysis, the existing analysis are presented under section 2.2. There are few related works in the research community, these are presented under section 2.3.

2.1 Spark Language

2.1.1 Spark Introduction

Ada language is used heavily in the safety and security critical systems. It is more suitable because of the separation of specification from the implementation and the parameter carry an extra qualification to specify the mode of use in the procedure. However, ada includes features which make static analysis difficult. Spark is a subset of ada language, which removes all the features of Ada that cause ambiguity in performing static analysis. Spark also includes interface specification language which allows the developer to specify contract, loop invariants, information flow, etc. The interface specification language is written as annotation within Ada comments block which allows spark programs to be compiler using existing Ada compilers.

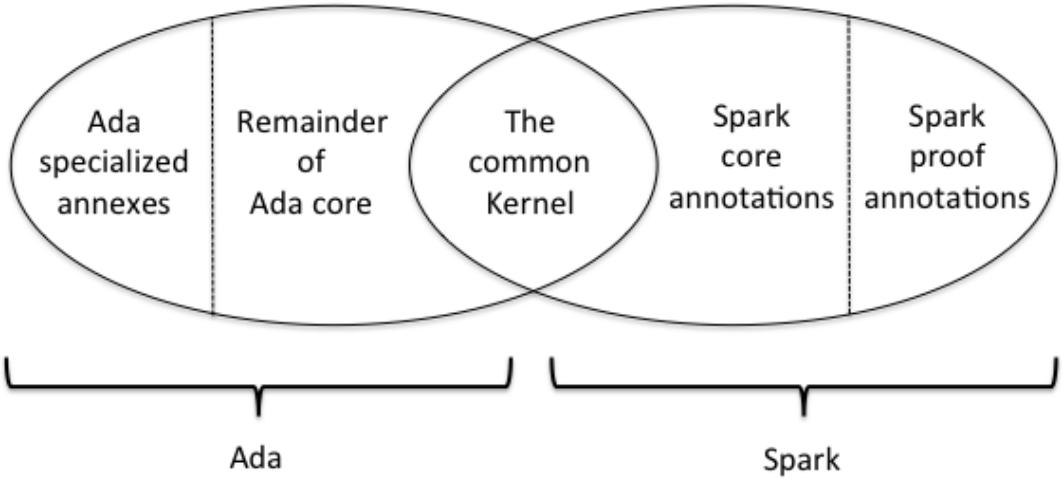

Figure 2.1: *Relationship between Ada and Spark*[10]

The figure 2.1 shows how spark language shares the kernel of Ada language and the additional features of spark. Spark uses only a portion of Ada which is considered safe and ignores features like recursive calls, statements that cause side effects, exceptions, generics, access types, and goto statements. Despite these omissions, the kernel language contains rich language features like package, private types, type extension, unconstrained array and functions returning composite types. With these features, powerful and safe programs can be developed.

2.1.2 Spark Interface Specification Language

Spark annotations are part of the specifications and the contract between subprograms. It is typically written before the actual implementation. Spark annotations are in two categories. The first category is flow analysis and visibility control, which comprises the core language and annotation. The second concerns formal proofs using additional optional annotations.

Spark allows two levels of analysis: data flow analysis which just checks the direction of the data flow, and information flow analysis, which considers the interaction between variables. The information flow analysis is optional. These analysis can be verified by the

Spark examiner developed by Praxis.

2.1.3 Features

```
1  package math is
2    procedure power (m: in Integer; n : in  Integer; result : out Integer);
3    --# derives result from m , n;
4  end math;
5  package body math is
6    procedure power(m: in Integer; n : in  Integer; result : out Integer)
7      is
8        loopvar : Integer;
9      begin
10       result := 1;
11       loopvar := n;
12       while (loopvar >= 1) loop
13         result := result * m;
14         loopvar := loopvar -1;
15       end loop;
16   end power;
17 end math;
```

Figure 2.2: *Spark program to compute power*

Figure 2.2 shows a Spark program which computes integer being raised to power. We can see two parts in the program, specification and implementation. The implementation part consists of a package encapsulating a procedure. In Spark, there can be multiple packages in a system and also multiple procedures and functions in a package. The parameters for the procedure carry an extra qualification specifying the direction of the information flow. Parameters which are qualified as 'in' can only be read from while the 'out' parameter should be written, a parameter with 'in out' mode should be read and written inside the procedure.

We can also see that the specification part is an abstraction of the implementation. In Spark there are two kinds of abstraction: abstraction by parameterization and abstraction by specification. In abstraction by parametrization the actual data being used is hidden. In the abstraction by specification the actual computation is hidden. The program show in 2.2 is an example of abstraction by specification, and the program in 4.1 is an example of

abstraction by parametrization.

The line 4 of the program described in figure 2.2 shows an annotation, which specifies 'result' get information only from 'm' and 'n'. This annotation is in the specification part of the program as there is no parameterized abstraction in the program 2.2.

```
1 procedure Add(X : in Integer);
2 --# global in out Total;
3 --# derives Total from Total, X;
```

Figure 2.3: *Spark global clause*[10]

Spark requires a global annotation to be used whenever the global variables are used inside a procedure. The example in the figure 2.3 shows the use of a global variable 'Total' where the flow is captured using derives clause.

```
1 procedure Inc;
2 --# global in out Total;
3 --# derives Total from Total;
4 --# post Total = Total~ + 1;
```

Figure 2.4: *Spark post clause*[10]

In the Figure 2.4, the post clause specifies the new value of 'Total' should be old value of 'Total' plus one, thus capturing the flow of information. The '~' symbol refers to the pre-state value of total.

2.2 Intra-procedural Analysis

Intra-procedural analyses are the analyses with their scope of operation limited to a single procedure. In other words, this analyses cannot perform when there is a procedure call, where the information flows from one procedure to another. The remaining of this section provides the basic definition and example for Control Dependence Graph, Reaching Definition, Data Dependence Graph, Program Dependence Graph and Intra-procedural slicing.

2.2.1 Graph Definitions

The intra-procedural analysis is performed on a graph, that is built to represent the given program. To understand the analysis, it is important to know the properties of a flow graph. Flow graph are directed graph, in which nodes represents the statements of a program and edges represents the a property that connects the two nodes.

Flow Graph

A flow graph $G = \{N, E, s, e\}$ consists of a set N of statement nodes, a set E of directed control-flow edges, a unique start node s with no incoming arcs, and unique end node with no outgoing arcs, such that all nodes in N are reachable from s, and e is reachable from all nodes in N[8].

Flow Graph Path

A *flow graph path π is a sequence of nodes $n_1 n_2..n_k$. Path π is said to be non-empty if it contains at least one node, and non-trivial if it contains at least two nodes. when the meaning is clear from the context, we will use π to denote the set of nodes contained in π . For example, we write $n \in \pi$ when n occurs in the sequence π*[8].

Node Domination

Node n dominates m in G (written dom(n, m)) if every path from the start node s to m passes through n (note that this makes the dominates relation reflexive). Node n strictly dominates node m in G if dom(n, m) and $n \neq m$[8].

Node Post-domination

Node n post-dominates node m in G (written post-dom(n, m)) if every path from node m to the end node e passes through n. Node n strictly post-dominates node m in G if post-dom(n, m) and $n \neq m$[8].

2.2.2 Control Flow Graph (CFG)

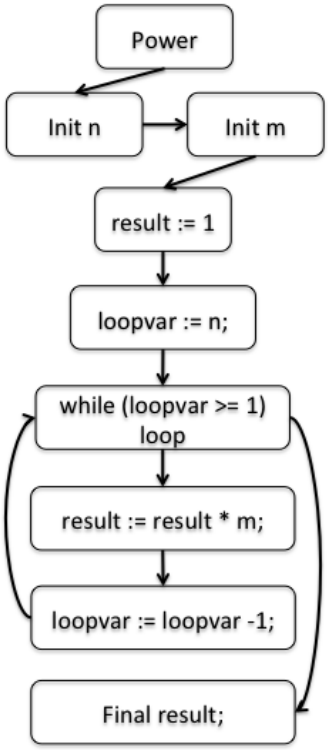

Figure 2.5: *Control Flow Graph for power program*

Control flow is the sequence of instructions in a program, ignoring the data values in register and memory, and the arithmetic calculations. A CFG is a graph with a separate node n for each assignment and jump in a given program p[8].

Figure 2.5 shows the CFG for the *power program* in figure 2.2. The node *power* is the start node and *Final result* is the end node. The *while* node contains two outgoing edges related to the two possible results from evaluating the loop conditional.

2.2.3 Reaching Definition

Definition

An assignment (called a definition in the classical literature) of the form $[x := a]^l$ *may reach a certain program point (typically the entry or exit of an elementary block) if there is an execution of the program where* x *was last assigned a value at* l *when the program point is reached.*

$$Kill_{RD} : Blocks_* \rightarrow P(Var_* \times Lab_*^?)^{12}$$

The Kill_{RD} function produces a set of pairs, where each pair consists of a variable and a label. For variable that are uninitialized we use the special label '?' and we set $\text{Lab}_*^? = \text{Lab}_* \cup \{?\}$. Each pair belongs to an assignments that is destroyed by the current block. An assignment is destroyed if the block assigns a new value to the variable.

$$gen_{RD} : Blocks_* \rightarrow P(Var_* \times Lab_*^?)^{12}$$

The gen_{RD} function produces a set of pairs of variables and labels of assignment generated by the block. Only assignments generate definitions.

l	$\textbf{Kill}_{\textbf{RD}}(l)$	$\textbf{gen}_{\textbf{RD}}(l)$
12	{(result,?), (result, 12), (result, 15)}	{(result, 12)}
13	{(loopvar, ?), (loopvar, 13), (loopvar, 16)}	{(loopvar, 13)}
14	∅	∅
15	{(result,?), (result, 12), (result, 15)}	{(result, 15)}
16	{(loopvar, ?), (loopvar, 13), (loopvar, 16)}	{(loopvar, 16)}

Table 2.1: *Reaching Definition analysis Gen and Kill for power program*

Figure 2.1 shows the *Kill* and *gen* sets for the *power program* in figure 2.2. We consider the line numbers of program as the labels as in the I column. At line number [12] the *result* is assigned to the value one. This assignment destroys all other assignments of *result*. So the *Kill* set for the label [12] consists of *result* associated with labels [?], [12 and [15] . The

l	$\mathbf{RD_{entry}}(l)$	$\mathbf{RD_{exit}}(l)$
12	{(n,*), (m,*), (result,?), (loopvar,?)}	{(n,*), (m,*),(result,12), (loopvar,?)}
13	{(n,*), (m,*), (result,12), (loopvar,?)}	{(n,*), (m,*), (result,12), (loopvar,13)}
14	{(n,*), (m,*),, (result,12), (result,15), (loopvar,13), (loopvar,16)}	{(n,*), (m,*), (result,12), (result,15), (loopvar,13), (loopvar,16)}
15	{(n,*), (m,*), (result,12), (result,15), (loopvar,13), (loopvar,16)}	{(n,*), (m,*), (result,15), (loopvar,13), (loopvar,16)}
16	{(n,*), (m,*), (result,15), (loopvar,13), (loopvar,16)}	{(n,*), (m,*), (result,15), (loopvar,16)}
Final result	{(n,*), (m,*), (result,12), (result,15), (loopvar,13), (loopvar,16)}	{(n,*), (m,*), (result,12), (result,15), (loopvar,13), (loopvar,16)}

Table 2.2: *Reaching Definition analysis: Entry and Exit set for power program*

gen is comprised of only the assignments made in this block which is the *result* assigned to one. Similarly *Kill* and *ten* are calculated for all the other statements in the *power program.*

The analysis is defined by the pair of functions RD_{entry} and RD_{exit} which map labels to a set of pairs of variables and labels of assignment blocks.

$$RD_{entry},\ RD_{exit}\ :\ Lab_* \rightarrow P(Var_* \times Lab_*^?)$$

The RD_{entry} and RD_{exit} are defined as

$$\begin{aligned} RD_{entry}(l) &= \begin{cases} \{(x,?) \| x \in FV(S_*)\} & if\ l = init(S_*) \\ \cup\{RD_{exit}(l') \mid (l',l) \in flow(S_*)\} & otherwise \end{cases} \\ RD_{exit}(l) &= (RD_{entry}(l) \setminus Kill_{RD}(B^l)) \cup gen_{RD}(B^l) \\ & \text{where} B^l \in \text{blocks}(S_*)\ ^{12} \end{aligned}$$

Figure 2.2 shows the entry and exit set computed for the power program. At the entry of the node [12], The m and n are parameters so the label is marked as '*' and other other variables are not defined at this point. In the exit set of [12] The result is defined at [12] is represented as (result,12). Similarly the extra and exist set are computed for the other labels.

2.2.4 Data Dependence Graph

A node n is *data-dependent* on node m if, for a variable v referenced at n, a definition of v at m reaches n. Thus, node n depends on node m because the assignment at m can influence a value computed at n.

Definition

Node n is data-dependent on m in program p (written $m \xrightarrow{dd} n$ *– the arrow pointing in the direction of the flow from m to n) if there is a variable v such that*

1. *there exists a non-trivial path* π *in p's CFG from m to n such that for every node* $m' \in \pi$ - $\{m,n\}$, $v \notin def(m')$, *and*
2. $v \in def(m) \cap ref(n)$[8].

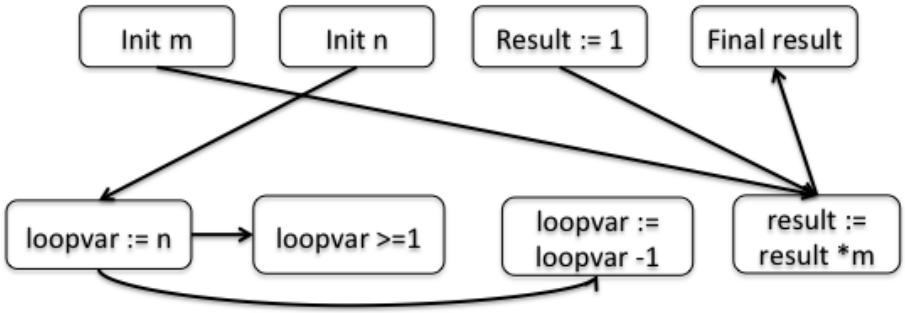

Figure 2.6: *Data Dependence Graph for power program*

In the power program of figure 2.2, *result* at [15] is data-dependent on [12] because the definition of result at [12] reaches the use of *result* at line [15]. This is denoted by the edge connecting from *result := 1* to *result := result *m* in the figure 2.6.

2.2.5 Control Dependence Graph

For a node to be *control-dependent* on m, m must have at least two immediate successors in the CFG, and there must be two paths that connect m with the unique end node such that one contains n and the other does not.

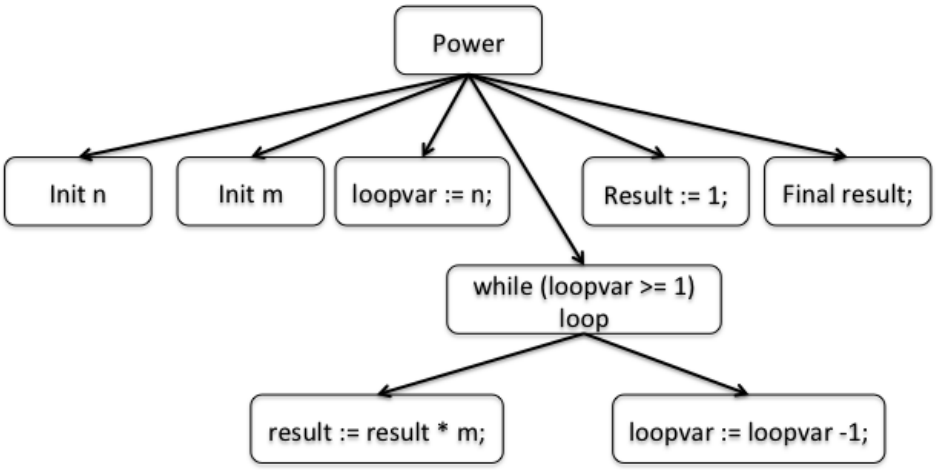

Figure 2.7: *Control Dependence Graph for power program*

Definition

Node n is control-dependent on m in program p (written $m \xrightarrow{cd} n$*) if*

1. *there exists a non-trivial path* π *from m to n in p's CFG such that every node* $m' \in \pi - \{ m, n\}$ *is post-dominated by n, and*

2. *m is not strictly post-dominated by n*[8].

In the power program 2.2, the statements [15] and [16] with respect to Figure 2.7 are *control-dependent* on the loop statement in the line [14]. The *Final result* is not control dependent on statement [14] since it post-dominates [14].

2.2.6 Program Dependence Graph

Given a program p,we define the relation $\rightarrow^{d}$ with respect to p to be the union of the CDG and the DDG with respect to p. The PDG P of p consists of nodes of the CFG G for p with edges formed by the relation $\rightarrow^{d}$[8].

$$\text{PDG} = \text{CDG} \cup \text{DDG}$$

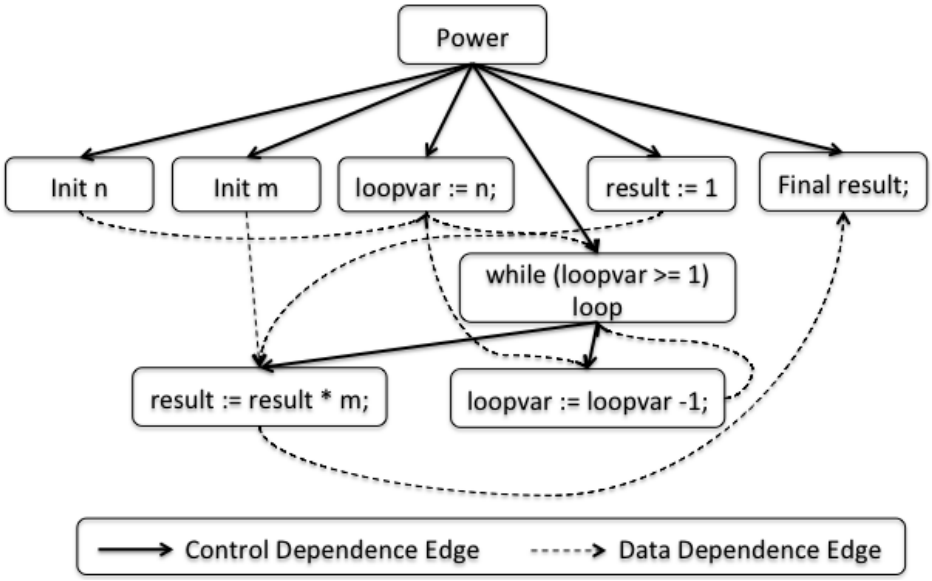

Figure 2.8: *Program Dependence Graph for power program*

Figure 2.8 shows the PDG of the power program. The node labeled *result := result *m* is control-dependent to the loop statement and it is data-dependent on the initial value of *m* and result.

2.2.7 Intra-procedural Slicing

Definition

The slice of a program with respect to program point p and variable x consists of all statements and predicates of the program that might affect the value of x at point p.

A slicing criterion is a pair <p, V>, where p is a program point and V is a subset of the program's variable.

The intra-procedural slice problem is simply a vertex reachability problem on a program dependence graph. Thus a slice of a program may be computed in linear time.

Slicing the power program with *Final result* as the slicing criterion will lead to the sliced program as seen in Figure 2.9. The form of slicing discussed thus far is termed a backward slice.

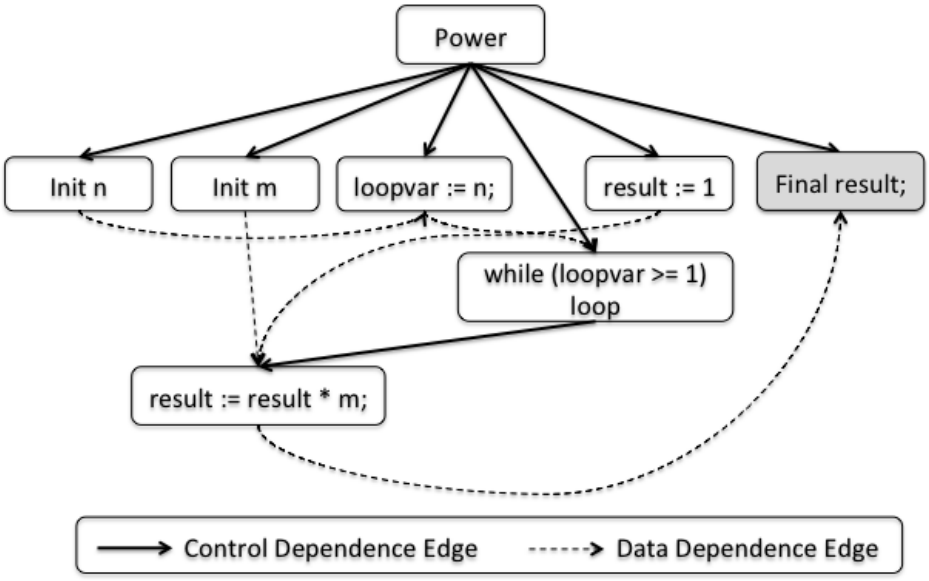

Figure 2.9: *Backward Slice of Power Program*

In figure 2.9 we can observe that a backward slice from *Final result* contains both the 'in' parameters m and n. For this program the slice from *Final result* is same as the original program as the *Final result* depends on all the nodes. The similar analysis is performed in the spark examiner by verifying the derives clauses. The only difference from the spark examiner is that this analysis provides a freedom of specifying the slicing criterion at any node within the procedure. This help the developer to infer the information flow than to verify the information flow contract. There is another variant of slicing called forward slicing which is used to compute the node that are receiving the information from the node under observation.

The forward slice on the power program with *init n* as the slicing criterion is shown in figure 2.10. The sliced set includes the *Final result*, which shows that the information flowing from *init n* reaches *Final result*.

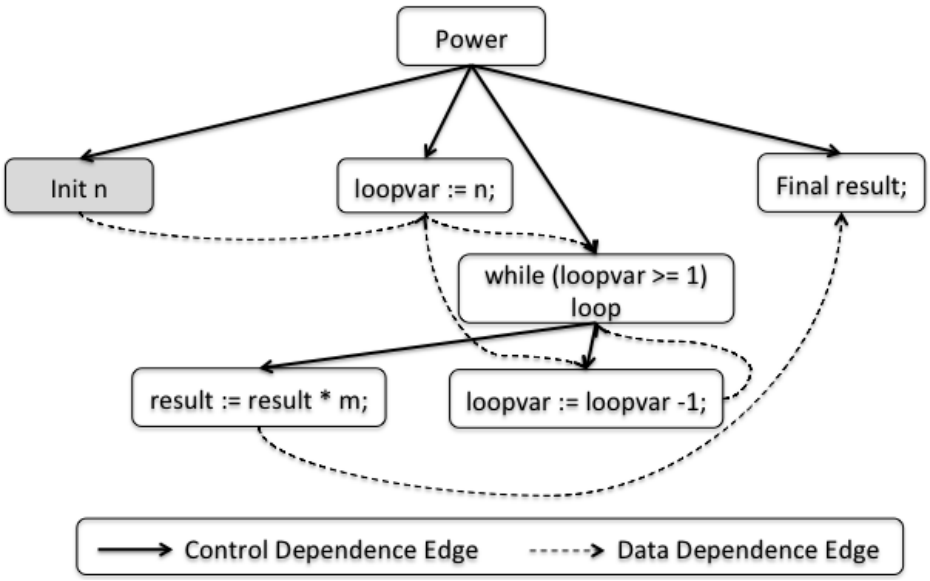

Figure 2.10: *Forward Slice of Power Program*

2.3 Information Flow Policy Language

2.3.1 JFlow

JFlow is an extension to the Java language that adds the ability to specify information flow annotations which can be statically checked. Features of JFlow include a decentralized label model, label polymorphism, run-time label checking, and automatic label inference. JFlow also supports many language features including objects, subclassing, dynamic type tests, access control, and exceptions.

In the decentralized label model, data values are labeled with security policies. A label is a set of security policies that restrict how a data value can flow through a program. Each policy in a label has an owner O, a set of readers which are principals that O allows to observe the data. A single principal may be the owner of multiple policies and may appear in multiple reader sets.

JFlow has access control that is both dynamically and statically checked. A method executes with a granted classification level. The authority is used to declassify some principal's data and also used to build more complex access control mechanisms. The expression *de-*

classify(e, L) relabels the result of an expression e with the label L. Declassication is checked statically, using the static authority at the point where the declassication is applied[11].

2.3.2 Java Information Flow

Java Information Flow(JIF) is a security-typed programming language that extends Java with support for information flow control and access control, enforced at both compile time and run time. Static information flow control can protect the confidentiality and integrity of information manipulated by computing systems. The compiler tracks the correspondence between information the policies that restrict its use, enforcing security properties end-to-end within the system. After checking information flow within JIF programs, the JIF compiler translates them to Java programs and uses an ordinary Java compiler to produce secure executable programs.

JIF extends Java by adding labels that express restrictions on how information may be used. For example, the following variable declaration declares not only that the variable x is an int, but also that the information in x is governed by a security policy:

```
1 \begin{center}
2     int {Alice → Bob} x;
3     \end{center}
```

In this case, the security policy says that the information in x is controlled by the principal Alice, and that Alice permits this information to be seen by the principal Bob. The policy {Alice ← Bob} means that information is owned by Alice, and that Alice permits it to be affected by Bob. Based on label annotations like these, the JIF compiler analyzes information flows within programs, to determine whether they enforce the confidentiality and integrity of information[2].

2.3.3 Information Flow Analysis of SPARK Examiner Tool

This tool was developed to improve the analysis provided by the Spark examiner. It provides features to mark state variables in package with different levels of security, which are specified

as constants variables. An example of such a classification can be seen in Figure 2.11

```
1 package Classify is
2          UNCLASSIFIED : constant := 0;
3          RESTRICTED : constant := 1;
4          CONFIDENTIAL : constant := 2;
5          SECRET : constant := 3;
6          TOPSECRET : constant := 4;
7 end Calssify;
```

Figure 2.11: *Specification of Security Classification*

This tool uses the Bell-LaPadula model of computer security, which enforces two properties.

1. no process may read data from a higher security level; and

2. no process may write data to a lower security level.

The verification of the security classification must agree on the above two properties.

Consider an example system with a *KeyStore* to store and manage a symmetric encryption key *SymmetricKey*, designed to mutate after every encryption according to the rotation parameter *RotorValue*. The variables are marked with security classification as follows.

The program in the figure 2.12 shows the *SymmetricKey* being classified as SECRET and the *RotorValue* classified as RESTRICTED. Thus the data from *SymmetricKey* cannot leak into *RotorValue*. The derives clause explicitly specifies the information flow of the program. The program can be checked using the information flow such that the integrity level of exported variable is no less than the integrity level of the import variable.

In the Rotate procedure the derives clause specifies *SymmetricKey* is derived from *RotorValue* and *SymmetricKey*. There is no leak in the program as the integrity level of *SymmetricKey* in export is equal or greater than the imports. The information flow at a procedure call is computed by substituting the actual parameters for formal parameters and checking for integrity flow[4].

```
1 --# inherit Classify;
2 package KeyStore
3 --# own SymmetricKey (Integrity => Classify.SECRET);
4 --# RotorValue (Integrity => Classify.RESTRICTED);
5 is
6    procedure Rotate;
7    --# global in RotorValue;
8    --#         in out SymmetricKey;
9    --# derives SymmetricKey from
10   --#          Symmetrickey, RotorValue;
11
12   procedure Encrypt (c : in MessageBlock;
13                      E : out MessageBlock);
14   --# global in SymmetricKey;
15   --# derives E from C, SymmetricKey;
```

Figure 2.12: *KeyStore Example to Show Information Flow Analysis*[4]

Chapter 3

Tool Architecture

This chapter presents an overall architecture of the information flow analysis tool. This tool is integrated to Eclipse, so a brief architecture of eclipse is presented under the section 3.1. This tool is built as a part of Sireum framework, the section 5.2 presents the architecture of Sireum.

3.1 Eclipse Architecute

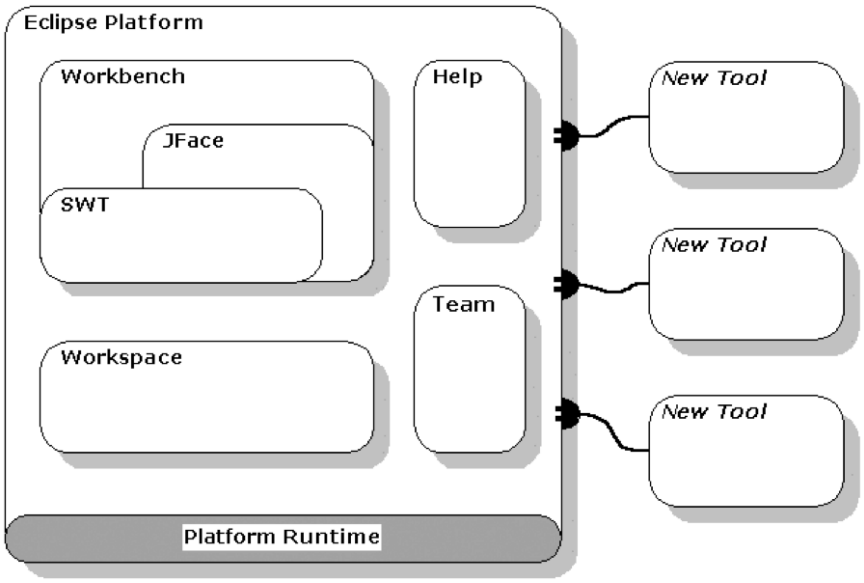

Figure 3.1: *Eclipse Platform Architecture*[9]

The Eclipse platform is designed for building integrated development environments (IDEs) that can be used to create applications targeting a wide range of domains. A plug-in is the smallest unit of the Eclipse Platform that can be developed and delivered separately. The Eclipse Platform itself is partitioned using the plug-in mechanism as shown in Figure 3.1, separate plug-ins provide the workspace, the workbench, and so on. Even the platform runtime itself has its own plug-in. A plug-in may declare any number of named extension points, and any number of extensions to one or more extension points declared in other plug-ins. A plug-ins extension point can be extended by other plug-ins. Also, any plug-in is free to define new extension points and to provide a new API for other plug-ins to use. Plug-ins are executed lazily, while executing a plug-in Eclipse uses the 'plug-in registry' to discover and access the extensions associated to it. The Eclipse Platform runs by a single invocation of a standard Java virtual machine. Each plug-in is assigned its own Java class loader that is solely responsible for loading its classes.

3.2 Sireum Architecture

Sireum is a software analysis platform developed at SAnToS Lab, Kansas State University. It incorporates various static analysis techniques such as data-flow framework, model checking, symbolic execution, abstract interpretation, and deductive reasoning techniques. It can be used to build various kinds of static analyzers aimed at various properties. Sireum means "ants" in the real Java language, i.e., Javanese.

3.2.1 Sireum Base

Base is the Sireum tooling framework. It provides a staged, parallel pipeline software component architecture, messaging system, as well as commonly used utility components.

Figure 3.2 shows the pipeline architecture of Sireum. It consists of a number of modules and clients. A client can be either a test suite or an Eclipse plug-in. For the client to

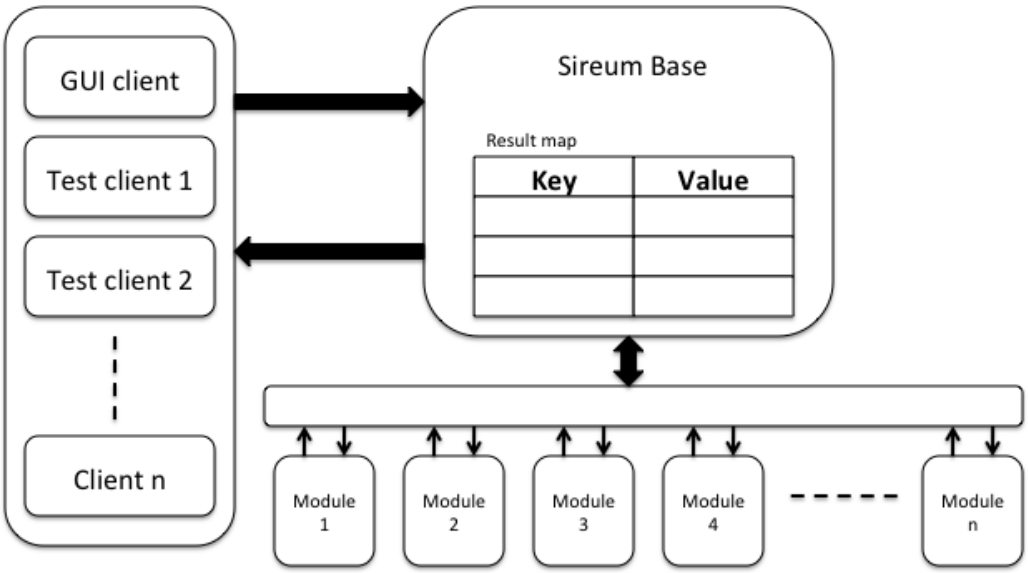

Figure 3.2: *Sireum Pipeline Architecture*

perform a specific analysis, the client passes a configuration file and input data to Sireum. The configuration file consists of the list of modules needed to perform the intended analysis. A module performs a specific analysis on the input and stores the result in the *result map*. This *result map* can be accessed by the next module in the pipeline. The modules can be executed in a serial or parallel manner based on a flag in the configuration file. At the end of the analysis, the client can access the result from the *result map*.

3.2.2 Pilar

Pilar is the Sireum modeling language. It serves as the single representation which all Sireum static analysis frameworks work on. It's extensible to allows modeling of a variety of system descriptions at different abstraction levels. The Spark program given as input is compiled to pilar before performing any kind of analyses.

3.2.3 Alir

Alir is the Sireum data-flow framework. It provides various configurable intra-procedural and inter-procedural analyses. The intra-procedural analyses are control flow analysis, reaching

definition analysis and data dependence analysis. These where the analyses provided by Alir before the implementation of the techniques discussed in this thesis. The inter-procedural analysis in Alir includes building, System dependence graph (SDG), slicing and chopping on SDG. Inter-procedural analysis is the contribution of this thesis, inter-procedural analyses are build on top of intra-procedural analyses framework.

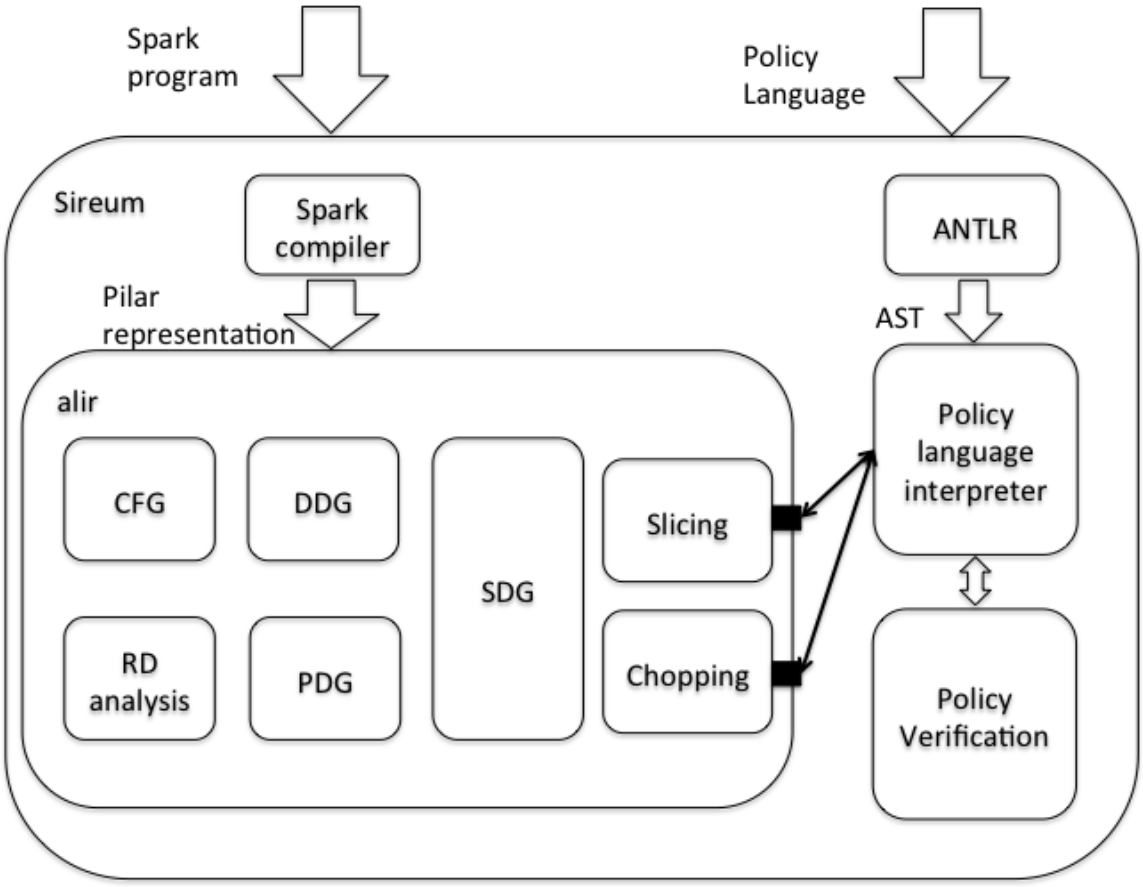

Figure 3.3: *Alir and Policy Language Architecture*

3.2.4 Policy Language

The verification of the policy language is a layer built on top of Alir. It interprets the policy, and then accesses Alir to compute the various slice and chop on the program as specified

in the policy. With the results from Alir, Sireum can then perform the verification of the policy against the program. The result of the policy language are displayed in a view as show in Figure 3.4, which is a part of the Eclipse plug-in for information flow analysis on Spark programs.

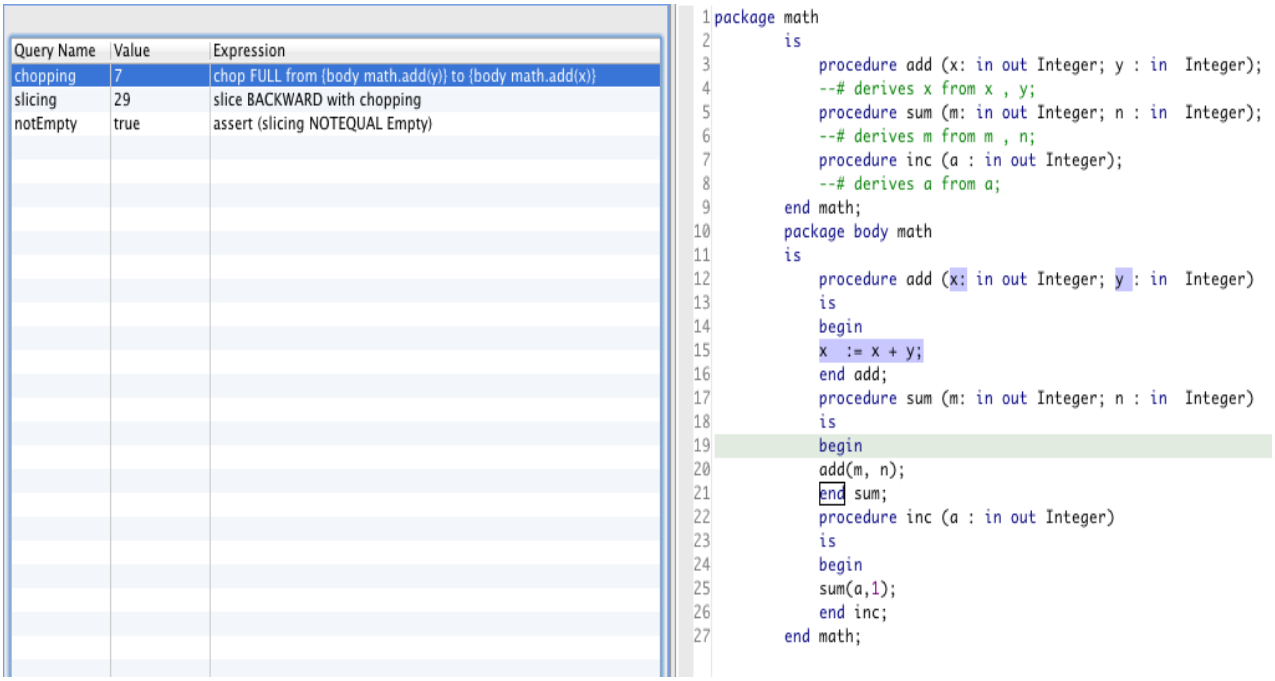

Figure 3.4: *Screen shot of policy view*

Chapter 4

System Dependence Graph

System Dependence Graph (SDG) is an extension of the program dependence graph defined in section 2.2.6 on page 15. SDG is union of multiple procedure's PDG and are linked by procedure calls.

A SDG includes PDGs, which represent the system's procedures, and some additional edges. The additional edges are of two types:

1. Edges that represent direct dependence between a call site and the called procedure, and

2. Edges that represent transitive dependence due to calls.

Extending the definition of dependence graphs to handle procedure calls requires representing the passing of values between procedures. In designing the representation of parameter passing, we have the following three goals

1. It should be possible to build an individual procedure's PDG with minimal knowledge of other system components.

2. The SDG should consist of a straightforward connection of the program dependence graphs.

3. It should be possible to extract a precise inter-procedural slice efficiently by traversing the graph.

The remainder of this chapter is organized as section 4.1 presents the parameter-in and parameter-out edges, there are the edges that connects two procedures to gather to form SDG. The section 4.2 presents summary edges, which are useful to get the information of a procedure call without entering in to the procedure. This is useful in slicing and chopping. The section 4.3 presents a technique to extend the SDG to support abstraction and refinement feature of Spark.

4.1 Parameter In - Parameter Out Edges

To meet the above goals, we follow a two-stage mechanism for runtime parameter passing. Consider procedures P and Q of a system. when P calls Q, values are transferred from P to Q by means of intermediate temporary variables, one for each parameter. The return values are transferred from Q to P using a different set of variables.

At the call site, procedure P copies the values of the actual parameters into the temporary variables, then procedure Q initializes its local variables from these temporary variables. Upon returning, procedure Q copies any return values into a set of temporary variable which are passed back to P. This model of parameter passing is represented in PDGs.

The PDGs are modified to contains 5 new vertices, A call site is represented using a *call-site* vertex. On the calling side, the information is transferred using *actual-in* vertices and received using *actual-out* vertices. These vertices are control dependent on the *call-site* vertex. The *actual-in* vertex contains an assignment statement that assigns the actual parameter to the temporary variable. Similarly the *actual-out* vertex contains an assignment from return temporary variables to the actual out parameters.

In the called procedure side, the information is received using *formal-in* vertex and transferred out using *formal-out* vertex. These vertices are control dependent on the procedure's entry vertex. Similar to the *actual-in* and *actual-out* vertices, *formal-in* and *formal-out* vertices contains assignment statements to transfer information to and from the temporary variables.

```
1 package p_simple_call
2 --# own AbsOwn, G;
3 --# initializes AbsOwn, G;
4 is
5   procedure R1(Q: in Integer; R: in out Integer);
6   --# global in AbsOwn; out G;
7   --# derives R from AbsOwn, R
8   --#         & G from Q;
9
10  procedure Q1(A: in Integer; B: in out Integer; C: out Integer);
11  --# global in AbsOwn; out G;
12  --# derives B from AbsOwn, B
13  --#         & C from AbsOwn, B
14  --#         & G from A;
15 private
16  G: Integer := 0;
17 end p_simple_call;
18
19 package body p_simple_call
20 --# own AbsOwn is X,Y;
21 is
22  X: Integer;
23  Y: Integer;
24
25  procedure R1(Q: in Integer; R: in out Integer)
26  --# global in Y; out G;
27  --# derives R from Y, R
28  --#         & G from Q;
29  is
30  begin
31    R := Y + R;
32    G := Q;
33  end R1;
34
35  procedure Q1(A: in Integer; B: in out Integer; C: out Integer)
36  --# global in X, Y; out G;
37  --# derives B from Y, B
38  --#         & C from X, Y, B
39  --#         & G from A;
40  is
41  begin
42    C := X + Y;
43    R1(A,B);
44    C := C + B;
45  end Q1;
46 begin
47   X := 1;
48   Y := 0;
49 end p_simple_call;
```

Figure 4.1: *Spark program to illustrate procedure call*

The *parameter-in* edge is the edge connecting *actual-in* to *formal-in* vertices. The *parameter-out* edge is the edge connecting *formal-out* to *actual-out* vertices.

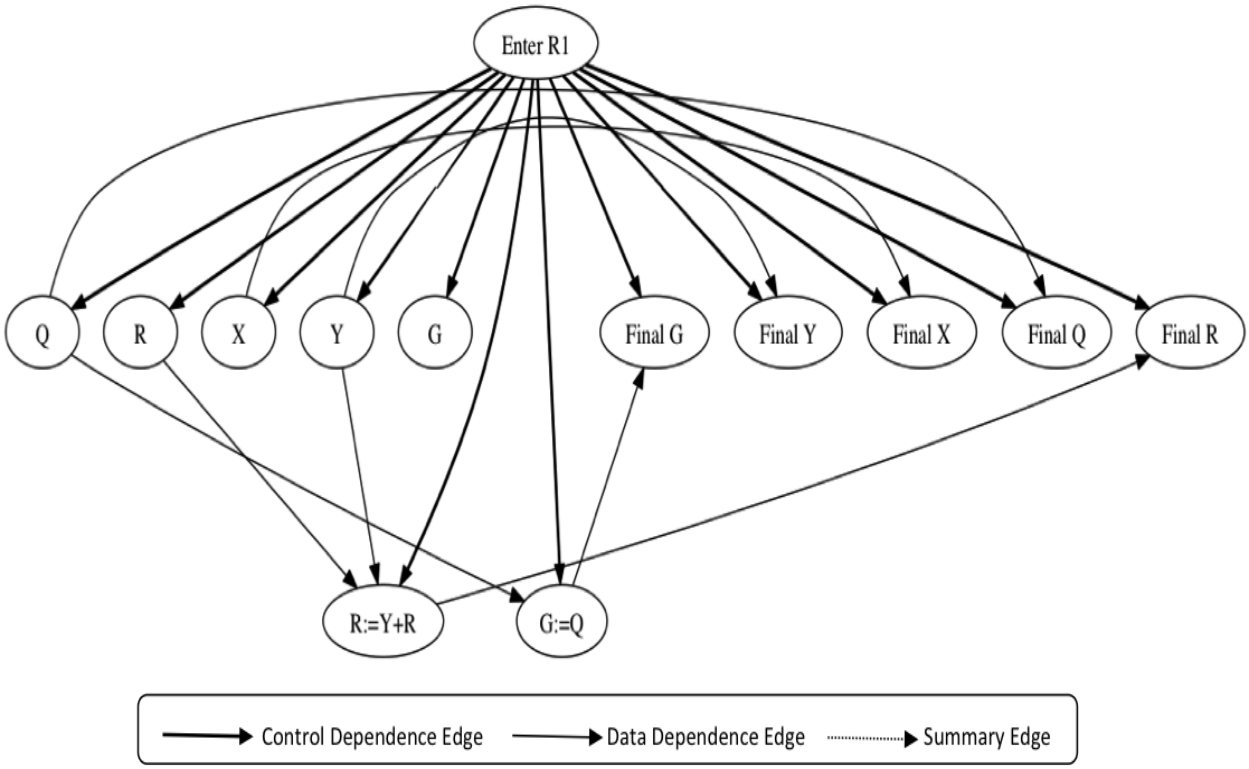

Figure 4.2: *Program Dependence Graph of procedure R*

4.2 Summary Edges

From the Spark derives clause it is easy to compute the dependency between *formal-out* vertices and *formal-in* vertices. From the *parameter-in* edge and *parameter-out* edge, given a *formal-in* vertex computing the corresponding *actual-in* vertex is simply the source of the *parameter-in* edge, similarly for a *formal-out* vertex, the target of the corresponding *parameter-out* edge which is a *actual-out* vertex. The dependencies between the *formal-out* and *formal-in* are extended to the corresponding *actual-out* and *actual-out* vertices. The edge connecting *actual-out* vertex to *actual-in* vertex are called summary edges.

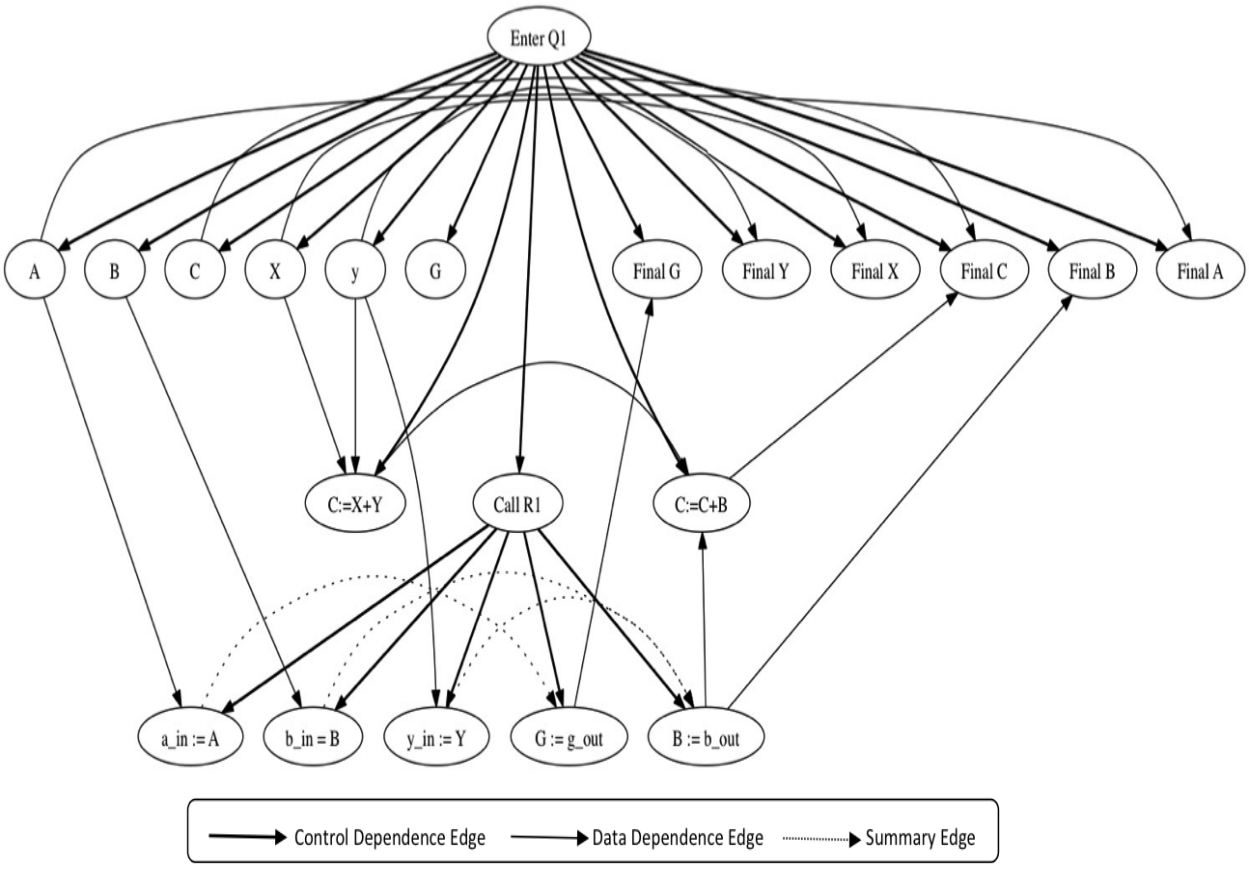

Figure 4.3: *Program Dependence Graph of procedure Q*

The program in figure 4.1 is a simple Spark program with procedure calls. On line number [43], we can notice the procedure Q1 calls R1, by passing A and B as arguments. This program also contains X, Y, and G as global variables. In SDG, in addition to the arguments, global variables are explicitly passed between procedures

Figure 4.4 shows the SDG of the program seen in 4.1. We can notice a *parameter-in* edge between $a\_in := A$ and $Q := a\_in$,*parameter-out* edge between $b\_out := R$ and $B := b\_out$, and summary edge between $b\_in := B$ and $B := b\_out$. Similarly all the *actual-in* and *actual-out* are connected to the corresponding *formal-in* and *formal-out* vertices. In this way all the parameters are linked and the two procedures are combined to form a single directed graph.

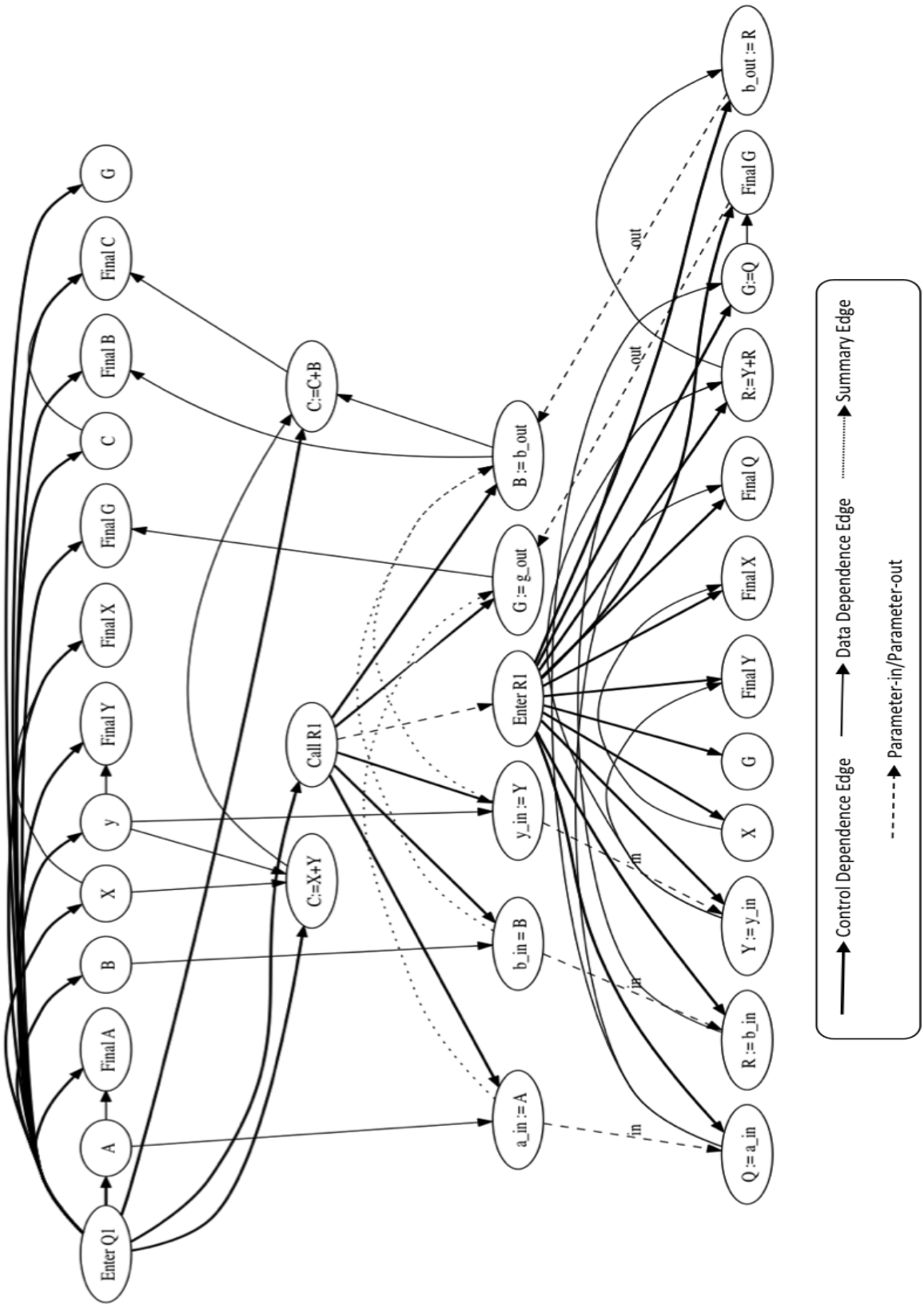

Figure 4.4: *System Dependence Graph*

4.3 Abstraction and Refinement

Spark's abstraction and refinement features has been incorporated in the SDG so that we can perform system level information flow analysis. To achieve this we introduce two new edges named *abs* and *ref*. The *ref* edge connects the initial parameters of specification and implementation. Any abstract parameter in the specification are refined to the corresponding initial parameters in the implementation. The *abs* edge connects the final parameters of implementation to the final parameters of the specification. The refined parameters are connected to the corresponding abstract parameters in the specification.

The figure 4.6 and 4.5 shows the PDGs of procedure Q1 and R1 with the abstract and refinement edges.

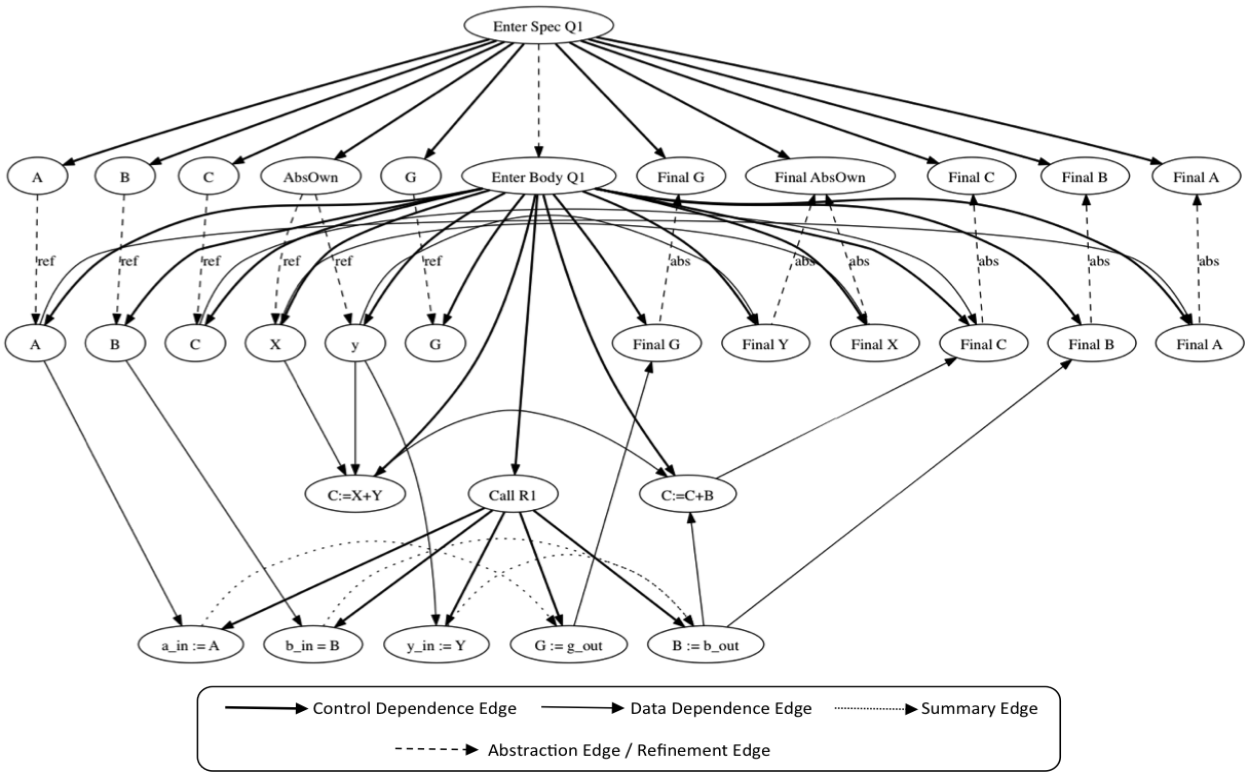

Figure 4.5: *PDG of Procedure Q1 with Abstraction and Refinement*

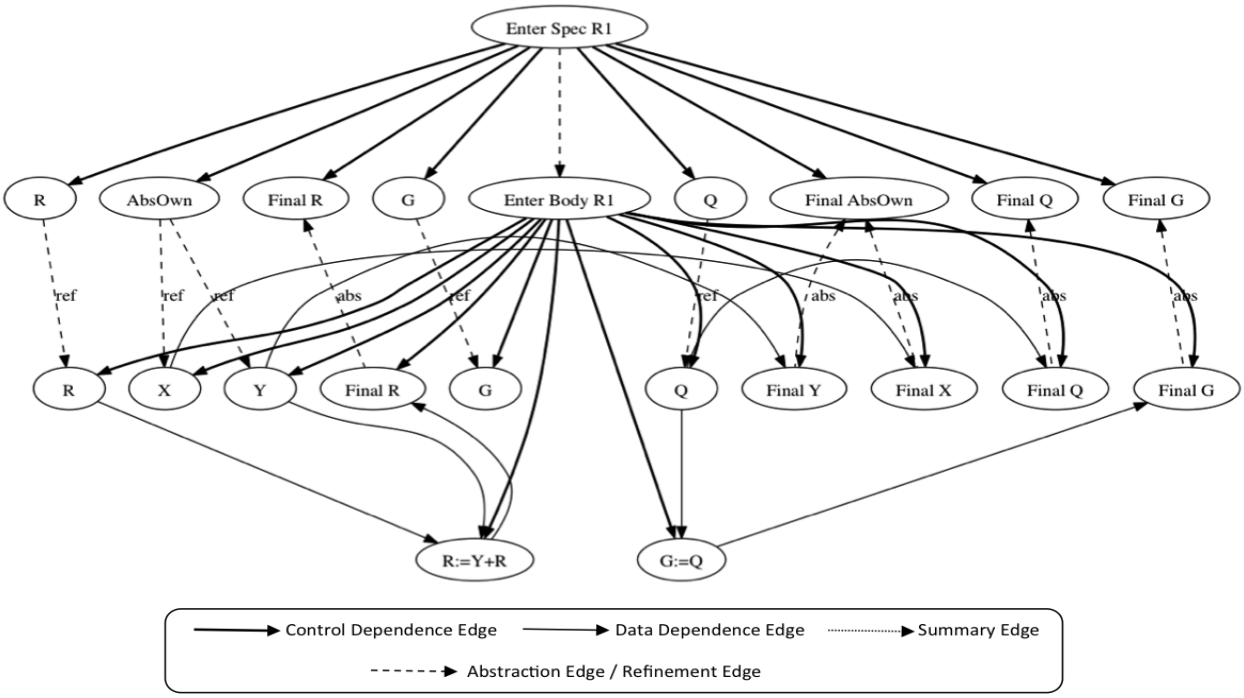

Figure 4.6: *PDG of Procedure R1 with Abstraction and Refinement*

4.3.1 Call Procedure Decision

Now for every procedure we have two PDGs. The PDG corresponding to the specification and the PDG corresponding to the implementation. Upon a call from a method Q to a method R, the determination of whether the information flows to the specification or the implementation depends on the location of R. If R is in the same package as that of Q then the *actual-in* and *actual-out* nodes are connected to the *formal-in* and *formal-out* nodes of the implementation. If R is not in the same package as Q then the *actual-in* and *actual-out* nodes are connected to the *formal-in* and *formal-out* nodes of the specification. By this we will follow the Spark examiners visibility rules.

In the figure 4.7 the SDG of program 4.1 is shown. In this graph the call to the procedure R1 is connected to the implementation of the R1 because both procedures belong to the same package. If they belonged to different packages then, the call to R1 would be conned to the specification of R1, Likewise the *parameter-in* and *parameter-out* edges would be connected to the formal parameters in the specification.

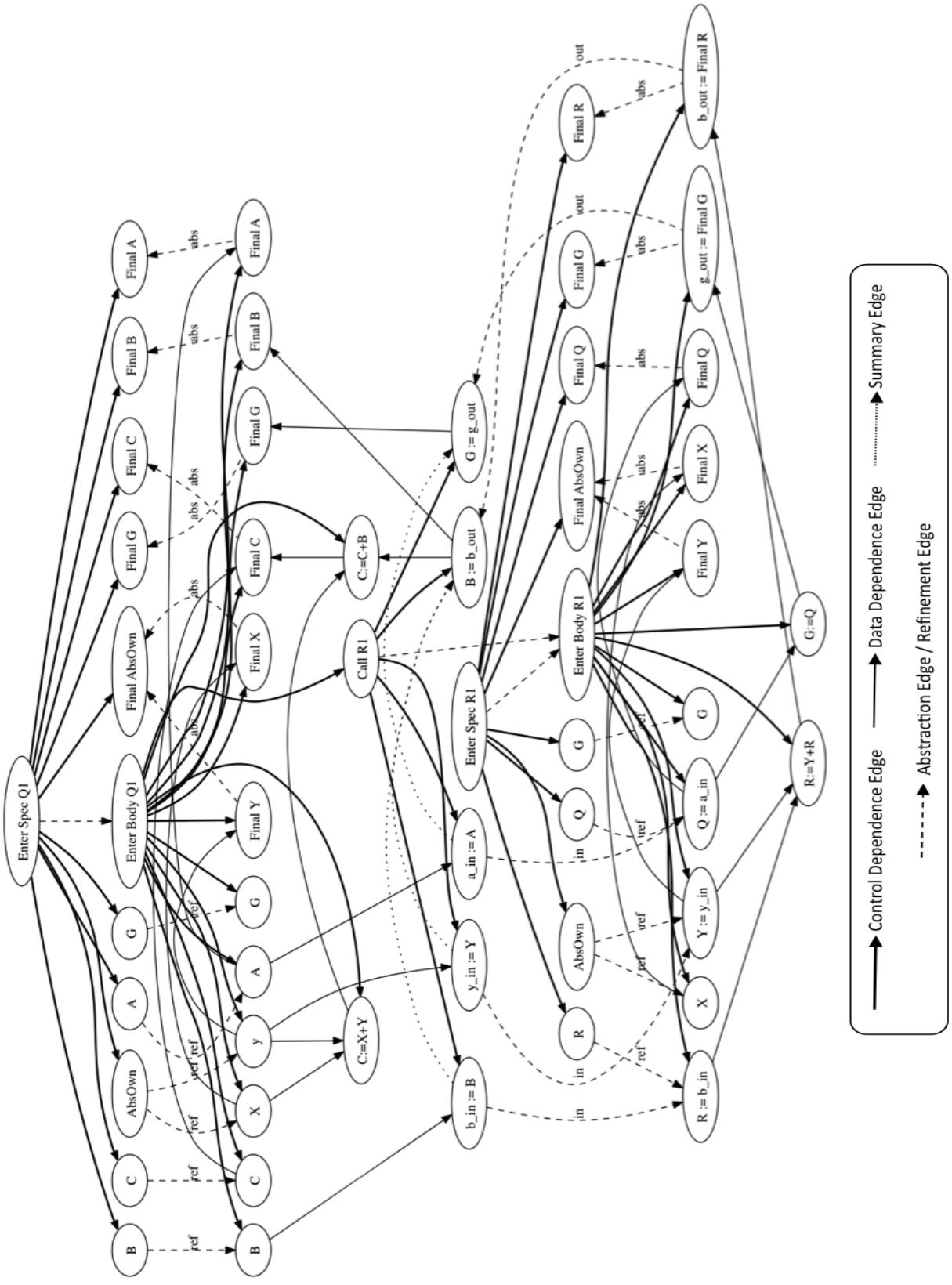

Figure 4.7: *SDG with Abstraction and Refinement*

Chapter 5

Inter-Procedural Slicing

Inter-procedural slicing is performed by traversing the SDG. The difficulty in inter-procedural slicing is keeping track of the calling context when a slice "descends" into a called procedure. The *summary* edges are the key element in solving this problem of tracing the call context. These edges represents data dependence from *actual-in* vertices to *actual-out* vertices due to procedure calls. These edges help us to slice without having to descend into a call. This remainder of this chapter is organizes as, section 5.1 presents a few basic definitions related to slicing. The section 5.2 and 5.3 presents backward and forward slicing.

5.1 Definitions

This section presents definition of slicing criterion, which is the input to perform slice operation. The slice set is the result of the slice operation. The slicing criterion is always part of the slice set.

5.1.1 Slicing Criterion

Slicing criteria is a set nodes based on which the information flow is computed. A slicing criterion C for a program p is a non-empty set of nodes

$$\{ n_1, \ldots . , n_k \}$$

where each n_i is a node in p's statement flow-graph[8].

5.1.2 Slice Set

Let C be a slicing criterion for program p and P be the graph for p. Then the slice set S_C of p with respect to C is defined as follows:

$$S_C = \{\ \mathrm{m} \mid \exists \mathrm{n}\ .\ \mathrm{n} \in \mathrm{C} \text{ and } \mathrm{m} \xrightarrow{d}{}^{*}\ \mathrm{n}\ \}^{8}.$$

5.2 Backward Slicing

The slicing of the SDGs are performed in two phases. The traversal in the first phase follows control dependent edges, data dependent edges, *parameter-in* edges, but not *parameter-out* edges. The traversal in phase two follows control dependent edges, data dependent edges, and *parameter-out* edges, but not *parameter-in* edges.

5.2.1 Phase I and Phase II

Consider a SDG G and a vertex s in procedure P, s being the slicing criterion. In order to slice on G with respect to s, in Phase I we identify vertices that can reach s, and are either in P itself or in a procedure that calls P. The procedure called by P are not in the slice because in Phase I, *parameter-out* edges are not followed. The summary edges are helpful in ignoring the called procedure by following the actual-in from the actual-out variables[6].

Phase II identifies vertices that can reach s from procedures called by P or from procedures called by procedures that call P. The phase II does not ascend into calling procedure because the *parameter-in* edges are not followed in phase II[6] .

5.2.2 Algorithm

The algorithm presented in Figure 5.1 consists of two phases. The slicing is performed in two phase because, the slice performed in single phase is imprecise. This algorithm will result in a precise backward slice for the given SDG and slicing criterion.

```
1 procedure MarkVerticesOfSlice(G,S)
2 declare
3   G: a system dependence graph
4   S, S': sets of vertices in G
5 begin
6   /* Phase 1: Slice without defending into call procedures */
7       MarkReachingVertices(G,S, {def-order, parameter-out})
8   /* Phase 2: Slice call procedure without ascending to call sites */
9       S' := all marked vertices in G
10      MarkReachingVertices(G, S', {def-order, parameter-in, call})
11 end
12 procedure MarkReachingVertices(G, V, Kinds)
13 declare
14   G: a system dependence graph
15   V: a set of vertices in G
16   Kinds: a set of kinds of edges
17   v, w: vertices in G
18   WorkList: a set of vertices in G
19 begin
20   WorkList :=V
21   while WorkList ≠ φ do
22     select and remove a vertex v from WorkList
23     Mark v
24     for each unmarked vertex w such that there is an edge w → v
25                           whose kind is not in Kinds do
26       Insert w into WorkList
27     od
28   od
29 end
```

Figure 5.1: *Two Phase Backward Slicing Algorithm*[6]

5.2.3 Example

The backward slice on SDG 4.4 is show in Figures 5.2 and 5.3 using the algorithm described in Figure 5.1 with slicing criterion $C := C + B$.

In the Figure 5.2, the node "B := b_out" is obtained by traversing the SDG from the slicing criterion "C := C + B". From the node "B := b_out", "b_out := Final R" is not reached because in Phase I out edges are not followed. It reaches the "y_in := Y" by following the summary edge, thus the rest of the slice is performed.

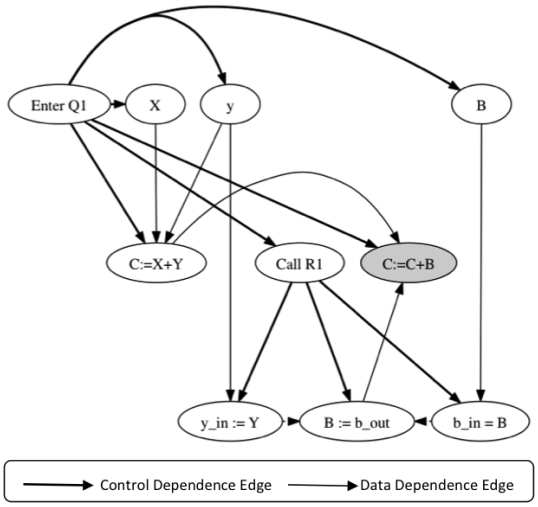

Figure 5.2: *Backward Slice Phase I of p_simpl_call Program*

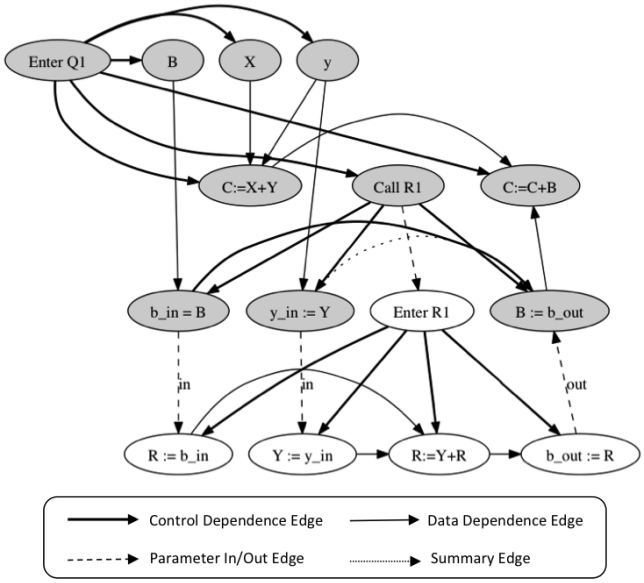

Figure 5.3: *Backward Slice Phase II of p_simpl_call Program*

5.3 Forward Slicing

5.3.1 Definition

The forward slice of a program with respect to a program point p and variable x consists of all statements and predicates of the program that might be affected by the value of x at point p.

Similar to the backward slicing, forward slicing is also performed in two phases. Phase I follows control dependent edges, data dependent edges, and parameter-out edges, but does not follow parameter-in edges. The traversal in phase I does not descend into called procedures because the parameter-in edges are not followed. The traversal in phase II follows control dependent edges, data dependent edges, and parameter-in edges, but not parameter-out edges. The traversal of Phase II does not ascend into calling procedures because the parameter-out edges are not followed.

5.3.2 Algorithm

```
 1 procedure MarkVerticesOfForwardSlice(G,S)
 2 declare
 3   G: a system dependence graph
 4   S, S': sets of vertices in G
 5 begin
 6   /* Phase 1: Slice forward without defending into called procedures */
 7       MarkVerticesReached(G,S, {def-order, parameter-in, call})
 8   /* Phase 2: Slice forward into called procedure
 9                  without ascending to call sites */
10       S' := all marked vertices in G
11       MarkVerticesReached(G, S', {def-order, parameter-out})
12 end
13 procedure MarkVerticesReached(G, V, Kinds)
14 declare
15   G: a system dependence graph
16   V: a set of vertices in G
17   Kinds: a set of kinds of edges
18   v, w: vertices in G
19   WorkList: a set of vertices in G
20 begin
21   WorkList :=V
22   while WorkList ≠ φ do
23     select and remove a vertex v from WorkList
24     Mark v
25     for each unmarked vertex w such that there is an edge v → w
26                            whose kind is not in Kinds do
27       Insert w into WorkList
28     od
29   od
30 end
```

Figure 5.4: *Two Phase Forward Slicing Algorithm*[6]

5.3.3 Example

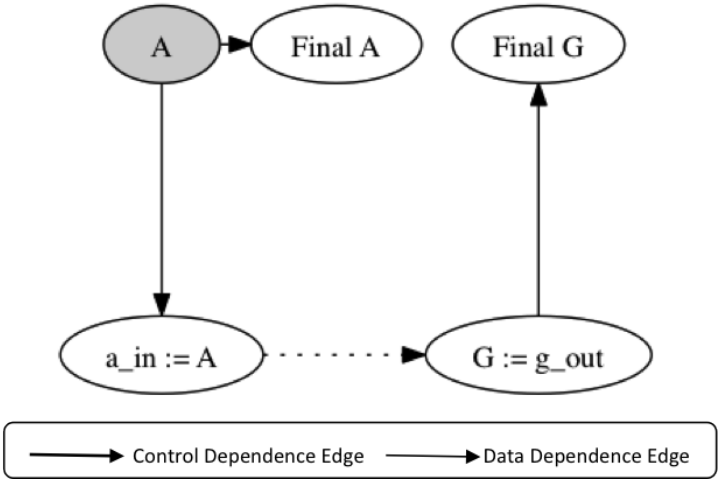

Figure 5.5: *Forward Slice Phase I of p_simpl_call Program*

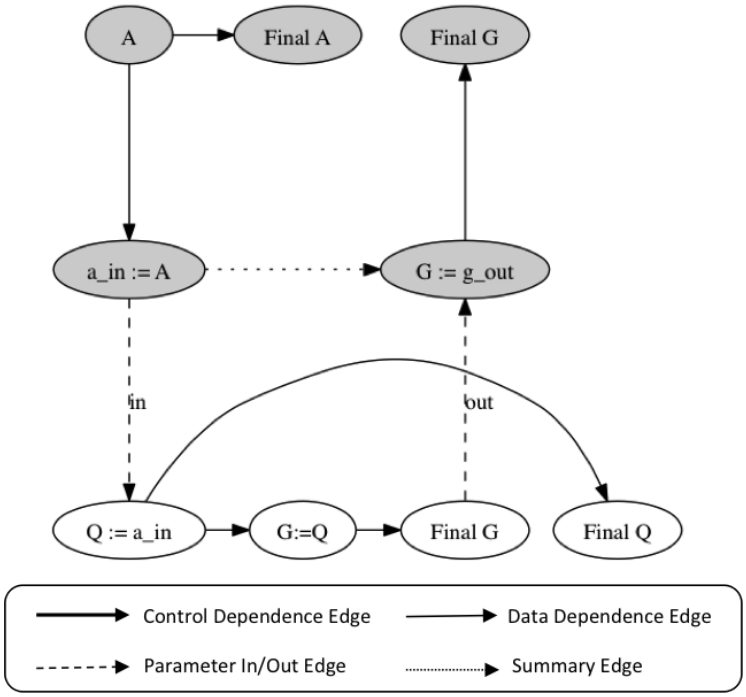

Figure 5.6: *Forward Slice Phase II of p_simpl_call Program*

The forward slice on SDG 4.4 is show in figures 5.5 and 5.7 using the algorithm described in Figure 5.1 with the slicing criterion *Init A*.

5.3.4 Screen Shot

The Figure 5.7 is the screen shot of the tool, when a backward slice is performed with slicing criterion as "C := C + B".

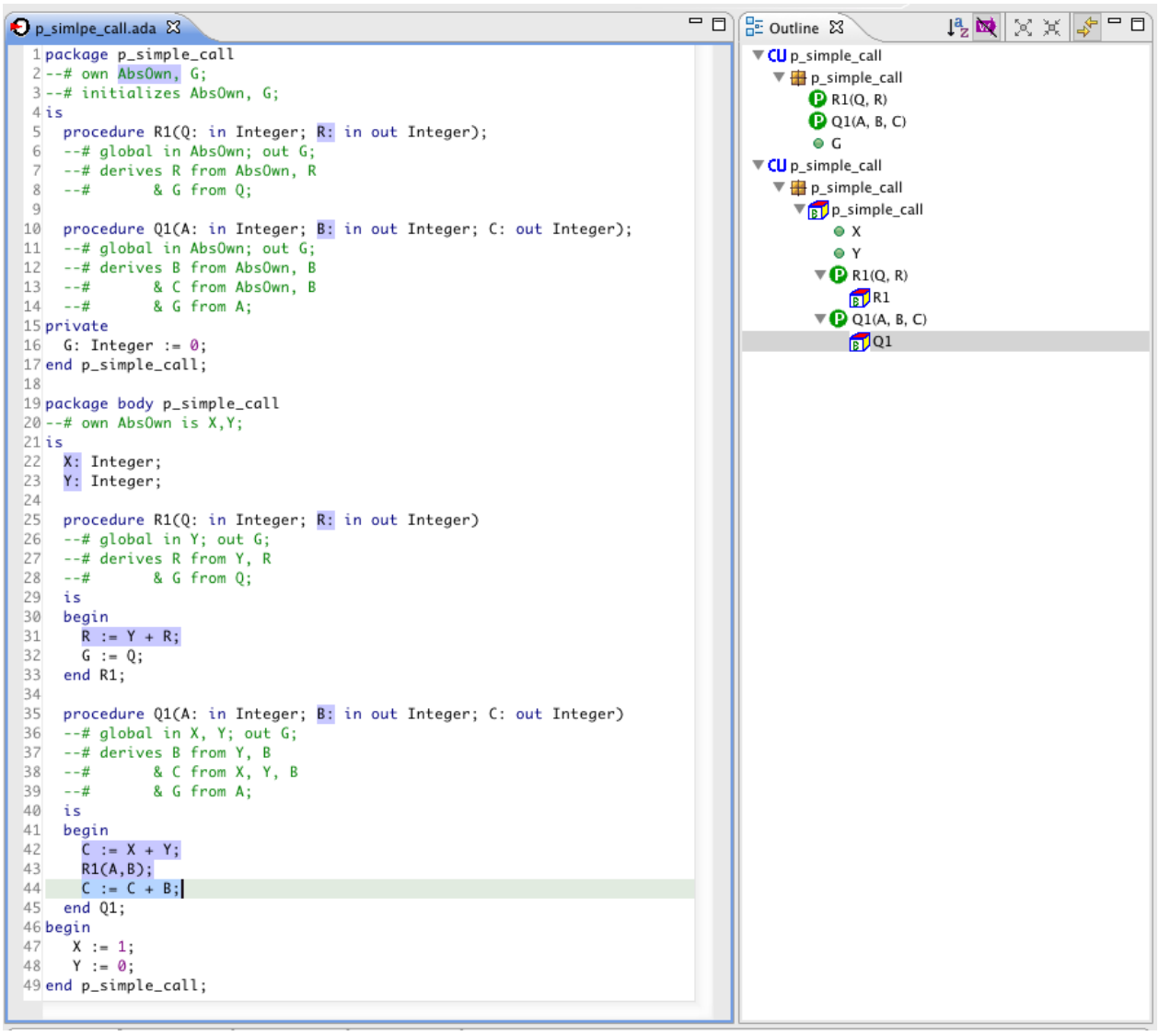

Figure 5.7: *Screen shot of backward slice*

Chapter 6

Inter-Procedural Chopping

Jackson and Rollings introduced a related operation to slicing called *program chopping*, which can be considered as a "filtered " slice. Chopping answers questions of the form "What are all the program elements v that serve to transmit effects from a given source element s to a given target element t?" Compered to slicing, chopping provides a more focused way of obtaining information about the transmission of effects through a program. An intra-procedural chop of a program from s to t is simply the intersection of forward slice from s and backward slice from t, provided s and t are in the same procedure.

$$\text{chop (s,t)} = \text{forward slice(s)} \cap \text{backward slice(t)}$$

Inter-procedural chopping involves generating a chop of an entire program, where the chopping is performed across procedure calls. By performing a chop on Add as seen in Figure 6 from the source "s = A : in out Integer" and target "t = B := B + 5", using the above definition, we get {x : in out Integer, x := x + y, s, t}. We can clearly see that there is no information flow between A and B, a precise chop would result in an empty set $\{\phi\}$.

This chapter presents a precise Inter-procedural chopping technique to overcome the errors involved with procedure calls. The remainder of this chapter is organized as follows, section 6.1 presents definitions, terminologies and notations. Section 6.2 presents the Inter-procedural chopping problems. Section 6.3 describes how to solve these chopping problem precisely. Section 6.4 presents Inter-procedural chopping algorithm with examples.

```
1 package Example is
2   procedure add ( x : in out Integer; y : in Integer ) ;
3     --# derives x from x,y;
4   procedure foo ( A : in out Integer; B : in out Integer ) ;
5     --# derives A from A
6     --# & B from B;
7 end example;
8
9 package body example is
10  procedure add (x : in out Integer; y : in Integer)
11  is
12  begin
13    x := x + y;
14  end add;
15  procedure foo (A : in out Integer ; B : in out Integer )
16  is
17    M : Integer;
18  begin
19    M := 3;
20    add(A,M);
21    add(B,5);
22    A := A + 2;
23    B := B + 5;
24  end foo;
25 end example;
```

Figure 6.1: *Example program to illustrate chopping*

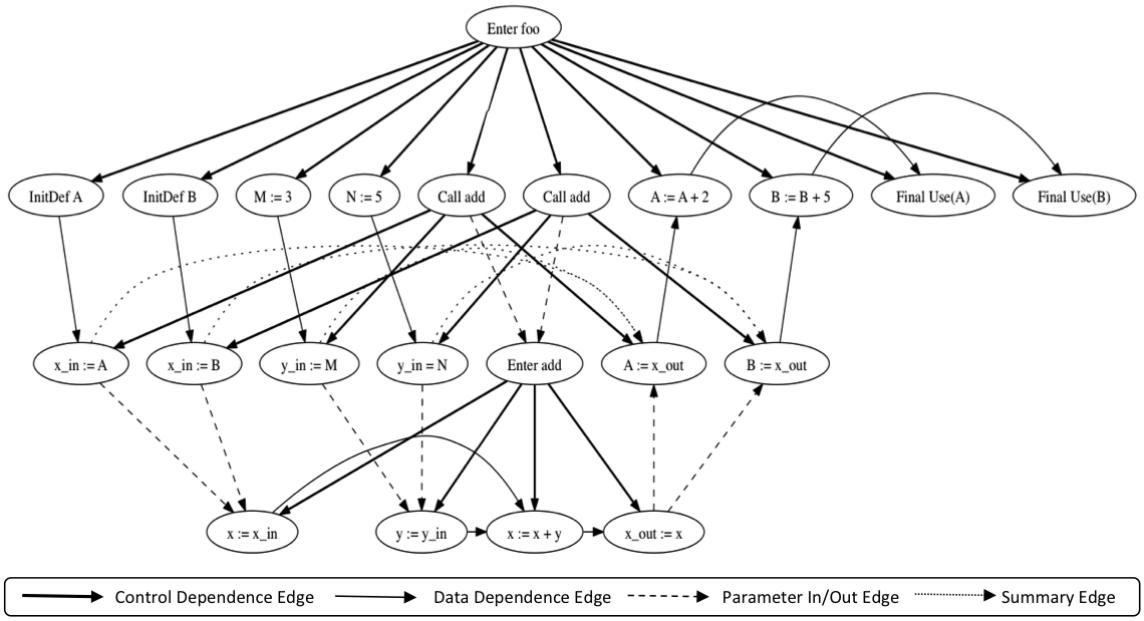

Figure 6.2: *System Dependence Graph for Chopping Program in 6.1*

6.1 Realizable Paths

Let each call vertex in SDG G be given a unique index from 1 to CSites, where CSites is the total number of call sites in the program. For each call site c_i, label the outgoing parameter-in edges and the incoming parameter-out edges with the symbols "$(_i$" and "$)_i$". Label all the other edges in G with the symbol e.

6.1.1 Same-level realizable path

A path in G is a *same level realizable path* if and only if the sequence of symbols labeling edges in the path is a string in the language generated from nonterminal*matched* by the following context-free grammar:

$$\begin{aligned} matched &\rightarrow matched\ matched \\ &\mid (_i\ matched\ )_i \qquad for 1 \leq i \leq CSites \\ &\mid e \\ &\mid \epsilon \end{aligned}$$

A same-level realizable path from node v to node w represents the transmission of an effect from v to w, where v and w are in the same procedure. A traversal of path from source to target, during which the call stack can temporarily grow deeper because of calls but never shallower than its original depth before eventually returning to its original depth[7].

6.1.2 Realizable Path

A path G is a realizable path if and only if the sequence of symbols labeling the edges in the path is a string in the language generated from nonterminal realizable by the following context-free grammar:

$$
\begin{array}{rcll}
\textit{unbalanced-right} & \rightarrow & \textit{unbalanced-right}\ )_i\ \textit{matched} & \textit{for}\ 1 \leq i \leq \textit{CSites} \\
& | & \textit{matched} & \\
\textit{unbalanced-left} & \rightarrow & \textit{unbalanced-left}\ (_i\ \textit{matched} & \textit{for}\ 1 \leq i \leq \textit{CSites} \\
& | & \textit{matched} & \\
\textit{realizable} & \rightarrow & \textit{unbalanced-right unbalanced-left} &
\end{array}
$$

A realizable path from v to w represents the transmission of an effect from v to w, where v and w are not required to be in same procedure. The "unbalanced-right" part of a realizable path represents an execution sequence that may leave the call stack shallower than it was originally, the "unbalanced-left" part represents an execution sequence that may leave the call stack deeper than it was originally[7].

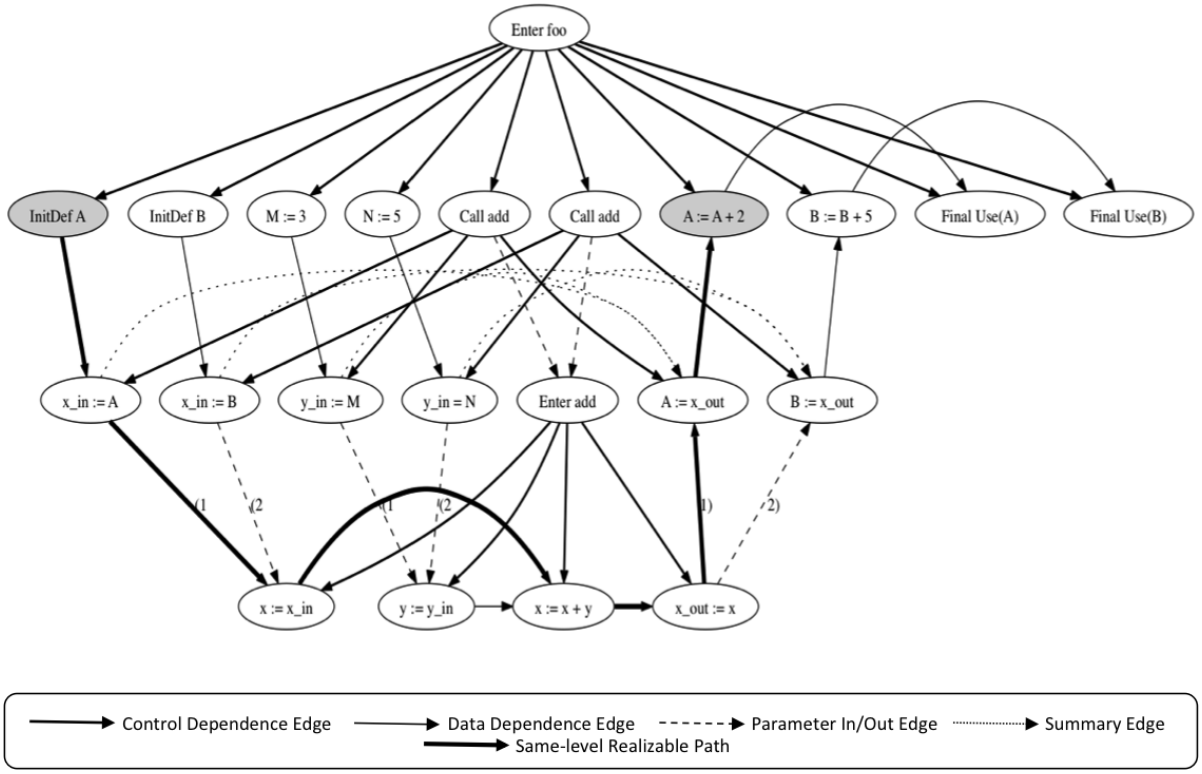

Figure 6.3: *Same-level Realizable Path for Chopping Program in 6.1*

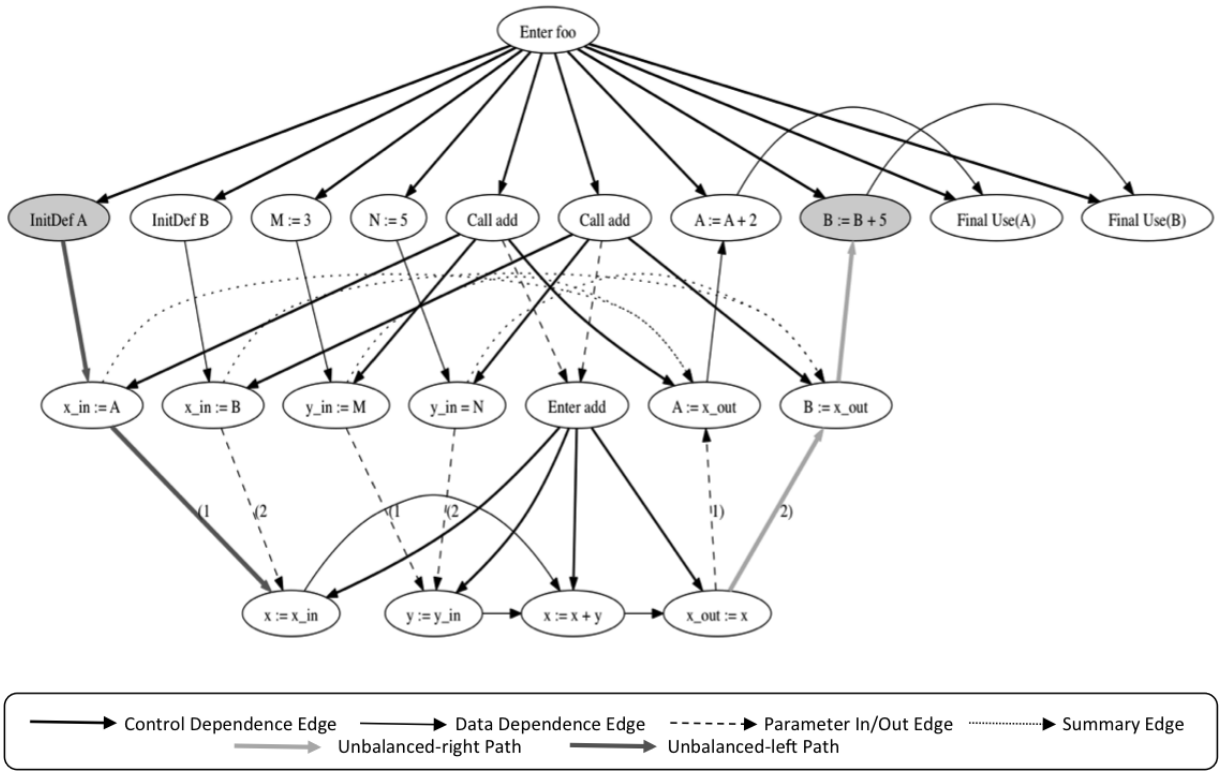

Figure 6.4: *Realizable Path for Chopping Program in 6.1*

6.1.3 Notations

The following notations are used in rest of this chapter:

$p = u \rightarrow^* w$	$:p$ *is a path from* u *to* w
$v \in p$	$:v$ *is one of the vertices of path* p
$u \rightarrow^*_m w$	$:matched$
$u \rightarrow^*_{unbr} w$	$:unbalanced - right$
$u \rightarrow^*_{unbl} w$	$:unbalanced - left$
$u \rightarrow^*_r w$	$:realizable$

6.2 Inter-procedural Chopping Problems

There four types of inter-procedural chopping :
In each of the following clauses, an SDG G is chopped with respect to given source vertex s and target vertex t.

unrestricted inter-procedural chop

consists of set of vertices given by :

$$\{\ v\ \mid \exists p\ such\ that\ p = s \rightarrow_r^*\ t\ and\ v\ \in\ p\}$$

This will perform a complete chop on all the cases i.e. source and target in same level and source and target in different levels.

Truncated unrestricted inter-procedural chop

consists of the set of vertices given by :

$$\{v \mid \exists\ w\ such\ that\ s\ \rightarrow_{unbr}^*\ v \rightarrow_{unbr}^*\ w \rightarrow_{unbl}^*\ t\}$$
$$\cup\{v \mid \exists\ w\ such\ that\ s\ \rightarrow_{unbr}^*\ w \rightarrow_{unbl}\ v \rightarrow_{unbl}\ t\}$$

This is a truncated chap in the sense it leaves out few edges corresponding the parameter call. Thus end up in an incomplete chop.

(non-truncated) same-level inter procedural chop

consists of the set of vertices given by:

$$\{v \mid \exists\ p\ such\ that\ p =\ \ s\ \rightarrow_m^*\ \ t\ and\ v\ \in p\}$$

This provide a complete chop between the source and target, provided the source and target should be in sam level.

Truncated same-level inter-procedural chop

consists of set of vertices given by:

$$\{v \mid s \rightarrow_m^* v \rightarrow_m^* t\}$$

This provides a truncated chop, when the source and target are in same level.

6.3 Precise Inter-procedural Chopping

In this section, we describe how to solve inter-procedural chopping problems precisely up to realizable paths. The summary edges of the SDG are defined as mentioned in the section 4.2. Once all the summary edges have been found it is easy to follow matched, unbalanced-right and unbalanced-left paths in the SDG.

$$f_m(S) =_{df} \{v \mid \exists s \in S \; such \; that \; s \rightarrow_m^* v\}$$
$$f_{unbr}(S) =_{df} \{v \mid \exists s \in S \; such \; that \; s \rightarrow_{unbr}^* v\}$$
$$f_{unbl}(S) =_{df} \{v \mid \exists s \in S \; such \; that \; s \rightarrow_{unbl}^* v\}$$
$$b_m(T) =_{df} \{v \mid \exists t \in T \; such \; that \; v \rightarrow_m^* t\}$$
$$b_{unbr}(T) =_{df} \{v \mid \exists t \in T \; such \; that \; v \rightarrow_{unbr}^* t\}$$
$$b_{unbl}(T) =_{df} \{v \mid \exists t \in T \; such \; that \; v \rightarrow_{unbl}^* t\}$$

Operations f_{unbr}, f_{unbl}, b_{unbr} and b_{unbl} correspond to the individual "slicing passes". All the six operations can be implemented by simple reachability computations on a SDG augmented with summary edges:

i The operation f_m(S) can be implemented as a reachability algorithm as follows: starting at members of S, the breadth-first search traverses control-dependence edges, flow-dependence edges, and summary edges, but not call edges, parameter-in edges, or parameter-out edges, and return all vertices encountered.

ii Operation f_{unbr}(S) can be implemented as a reachability algorithm that starts at members of S and traverses control-dependence edges, flow-dependence edges, summary edges, and parameter-out edges, but not call edges or parameter-in edges.

iii The operation b_{unbl}(T) can be implemented as a backwards reachability algorithm: starting at members of T, it traverses control-dependence edges, flow-dependence edges, call edges, summary edges, and parameter-in edges, but not parameter-out edges.

The other three operations are implemented similarly.

6.4 Algorithm

6.4.1 Truncated Same level Chop

```
1 function TruncatedSameLevelChop(s, t) returns vertex set
2 begin
3   return f_m({s}) ∩ b_m({t})
4 end
```

Figure 6.5: *Algorithm for Truncated Same-level Chop*[7]

Truncated same-level chop is applied when the source and target are in the same procedure. In the algorithm shown in Figure 6.5 the $f_m(\{s\})$ is calculated by a forward slice from the source without following parameter-in edges, parameter-out edge and call edges. Similarly the $b_m(\{t\})$ is computer by a backward slice on the target without following parameter-in edges, parameter-out edge and call edges. The intersection of the two $f_m(\{s\})$ and $b_m(\{t\})$ gives the result.

For example, Consider source s = *InitDef A* and target $t = A := A + 2$. The Truncated same-level chop will result in {*InitDef A, x_in* := *A, A* := *x_out, A*:= *A +2*}.

6.4.2 Same levelChop

Consider the same Source and target as $s = InitDef\ A$ and $t = A := A + 2$. After performing Truncated same-level chop there exists a summary edge from $x\_in := A$ to $A := x\_out$ in the result. In the same-level chop Aux the formal-in and formal-out related to the summary edge are considered to be source and target. With this new source and target, Truncated same-level chop is performed and the results are aggregated. Thus the truncated same-level chop is performed for all the summary edge appear in the result. By this technique all the procedures involved in the information flow between source and target are explored and the completed chopped is obtained, provided the source and target are in same level.

6.4.3 Truncated Chop

The Figure 6.9 shows the result of "W" referred in algorithm 6.8 by the intersection of forward unbalanced right and backward unbalanced left on source $x:=x\_in$ and target $B:= B+5$. Note that the source and target are in different different procedures. From the computed W, the vertices to the right of W are computed by intersecting forward unbalanced right of the source and backward unbalanced right of W. Similarly the vertices to the left of W are computed by intersecting forward unbalanced left with W and backward unbalanced left with target. The truncated chop is obtained by the union of right side vertices and left side vertices.

6.4.4 Non-truncated Chop

A non-truncated chop is computed by first performing a truncated chop in order to balance the source and target. If there are any summary edges in the result of Truncated Chop, the Same-level Chop Aux is performed to explore all other procedures calls present in the result. The procedures called by the source or target are not fully explored by a truncated chop.

```
1 function SameLevelChop(s, t) returns vertex set
2 declare
3   S: vertex set
4   WorkList: set of summary edges
5 begin
6   S := TruncatedSameLevelChop(s, t)
7   WorkList := {(x, y) ∈ SummaryEdges | x, y ∈ S}
8   return SameLevelChopAux(S, WorkList)
9 end
```

Figure 6.6: *Algorithm for Same-level Chop*[7]

```
1 function SameLevelChopAux(Answer, WorkList) returns vertex set
2 parameters
3   Answer: vertex set
4   WorkList: set of summary edges
5 declare
6   S: vertex set
7 begin
8   Remove all marks on {formal-in, enter} formal-out pairs
9   while workList ≠ φ do
10    select and remove a summary edge(v, w) from WorkList
11    let v' = the formal-in or enter vertex that corresponds to v and
12        w' = the formal-out vertex that corresponds to actual-out w
13    in
14      if (v', w') is unmarked then
15        Mark(v', w')
16        S := TruncatedSameLevelChop(v', w')
17        Answer := Answer ∪ S
18        WorkList := WorkList ∪ {(x, y) ∈ SummaryEdges | s, y ∈ S}
19      fi
20    ni
21  od
22  return Answer
23 end
```

Figure 6.7: *Algorithm for Same-level Chop Aux*[7]

```
1 function TruncatedChop(s, t) returns vertex set
2 declare
3   W, VR, VL: vertex set
4 begin
5   W := f_unbr({s}) ∩ b_unbl({t})
6   VR := f_unbr({s}) ∩ b_unbr({W})
7   VL := f_unbl({W}) ∩ b_unbl({t})
8   return VR ∪ VL
9 end
```

Figure 6.8: *Algorithm for Truncated Chop*[7]

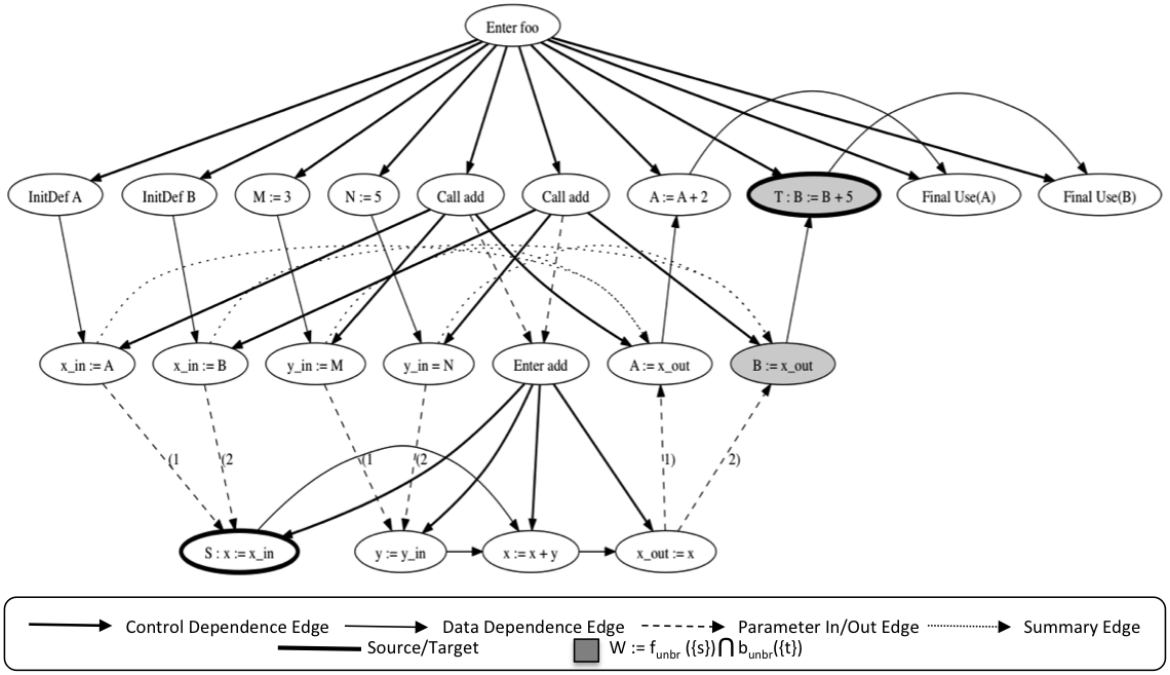

Figure 6.9: *W for Chopping Program in 6*

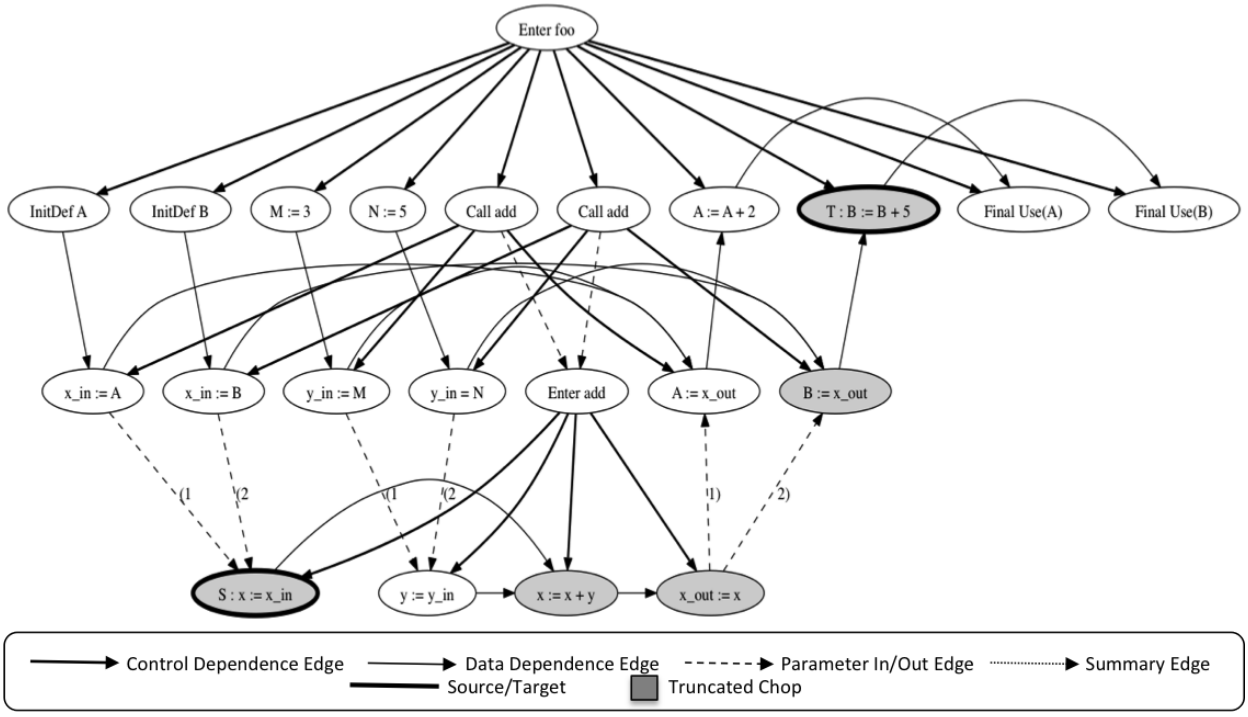

Figure 6.10: *Truncated Chop for Chopping Program in 6*

```
1 function Chop(s, t) returns vertex set
2 declare
3   W, VR, VL: vertex set
4   WorkList: set of summary edges
5 begin
6   /*Perform a TruncatedChop*/
7   W := f_unbr ({s}) ∩ b_unbl({t})
8   VR := f_unbr ({s}) ∩ b_unbr({W})
9   VL := f_unbl ({W}) ∩ b_unbl({t})
10  /* Invoke SameLevelChopAux with all summary edges on
11   * unbalanced-right paths from s to W or unbalanced-left
12   * paths from W to t */
13  WorkList := {(x, y) ∈ SummaryEdges | x, y ∈ VR}
14              ∪ {(x, y) ∈ SummaryEdges | x, y ∈ VL}
15  return SameLevelChopAux(VR ∪ VL, WorkList)
16 end
```

Figure 6.11: *Algorithm for Non-truncated Chop*[7]

A non-truncated chop completely explores every path that can contribute information from source to target.

Chapter 7

Policy Language

This chapter presents a language that represents the policy language. We use this language to infer information flow propertiesand to specify and verify security policies. The section 7.1 presents the grammar of the policy language and explains the features with examples. Section7.2 provides the verification technique used check the security policies.

7.1 Grammar

The Grammar of the policy language is classified into four blocks: query block, assert block, policy block and check block.

```
Compilation  ::= query_block
               | assert_block
               | policy_block
               | check_block
```

7.1.1 Policy Block

```
policy_block ::= policy ID:
                 policy_E;

policy_E     ::= domain dom_set
                 (order dom_order)?

dom_set      ::= {ID*}

dom_order    ::= ID ( <= ID )*
```

The policy block is used to specify the security classification domain and the partial ordering between various domains. If the order is not provided between some domains, then they are treated as disjoint domain.

Policy Language Example

```
policy Security :
  domain {secret, classified, public}
  order public <= classified <= secret;
```

In the above policy, secret, classified and public are the domains and the partial ordering indicated that public has the lowest priority and secret has the highest priority.

7.1.2 query block

```
query_block  ::= query ID:
                 E;

E            ::= E1 union E2
               | E1 intersection E2
               | E1 minus E2
               | slice_exp
               | chop_exp
               | ( E )
               | qry_name
               | bool_literal
               | empty_litral
               | litral_set

slice_exp    ::= slice (forward | backward)? with E

chop_exp     ::= chop (fast | truncated | samelevel | full)? from E1 to E2

bool_litral  ::= true | false

empty_litral ::= empty

litral_set   ::= {args*}

args         ::= label
               | proc_name

label        ::= <<ID>>

proc_name    ::= (proc_type)? path ( param* )
```

```
proc_type     ::= spec | body

path          ::= ID (. ID)*

param         ::= param_type (path | ID)

param_type    ::= in | out
```

Query block is used to infer the information flow properties. The result of each query block is a set of nodes, which are extracted from Spark Ada programs by slicing or chopping on procedure parameters or specifying the label to identify a particular statement. The query name acts as a variable to hold the result, which can be used in later blocks.

```
query chopping:
  chop full from {math.add(in y)} to {math.add(out x)} ;
```

The above query performs a chop from the *Init y* to *Final x* in add procedure and storers the result in the variable *chopping*.

```
query slicing:
  slice backward with chopping;
```

This query performs a backward slice with the result of the previous query as the slicing criterion.

7.1.3 Check block

```
check_block   ::= check ID:
                  check_E;

check_E       ::= check ID with provided node_class+ required node_class+

node_class    ::= E : ID
```

Check block is used to verify the specified security policy. If no error is found during verification then it returns true and an empty set. If the verification fails it returns false and the chop of the path contributing to the information leakage, which is a set of nodes.

```
check case1:
  check Security with provided slicing: public required chopping : secret;
```

This verify the policy Security with the slicing and chopping sets. The slicing is classified to public and the chopping is classified to secret.

7.1.4 Assert Block

```
assert_block ::= assert ID:
                 assert_E;

assert_E     ::= assert (E1 (= | /= | subsetof | supersetof | disjointof) E2)
```

Assert block is used to compare the two information flows. It also provides set comparison operators like subset, superset and disjoint. The result of assert block is always a boolean.

```
assert notEmpty:
  assert (slicing /= empty);
```

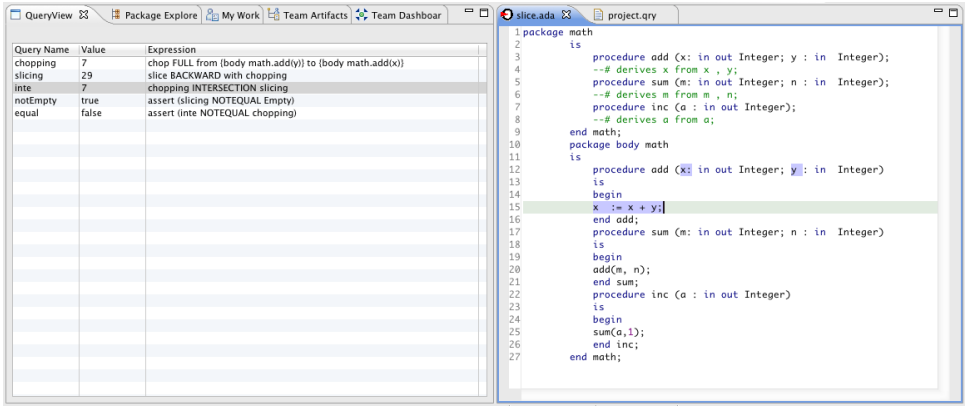

Figure 7.1: *Screen shot of query results*

7.2 Policy Verification

Using the features of Policy Language we can specify information flow control policies. In this section we will discuss the verification technique used to check a specified policy against Spark Ada programs.

7.2.1 Security level

The theoretical models of information flow control utilize a lattice L = (L, ⊔, ⊓) of security levels, the simplest consisting of two security levels *High* and *Low*. This approach of verifying

the security policy is based on dependence graphs. If there is no SDG path from a to b, it is guaranteed that there is no information flow between a and b.

The security level of a statement with SDG node x is written S(x), and confidentiality requires that an information receiver must have at least the same security level of any sender. In SDG, this implies $\forall y \in pred(x)$: $S(x) \geq S(y)$ which ensures $S(y) \rightsquigarrow S(x)$. The dual condition for integrity is $\forall y \in pred(x)$: $S(x) \leq S(y)$. However, this assumes that every statement representation node has a security level specified, which is not true in reality. We want to specify provided and required security levels for only certain nodes that we are interested in and not for all nodes.

The provided security level specifies that a statement sends information from a particular security level and the required security level specifies that only information with a lower security level may reach that statement. The provided security levels are defined by a partial function P: $N \rightarrow L$, where N is the set of nodes representing statements of the programs. Thus, $l = P(s)$ specifies the statement's security level.

The required security levels are defined similarly as a partial function R: $N \rightarrow L$. Thus, P(s) specifies the security level of the information generated at s and R(s) specifies the maximal allowed security level l of the information that can be reached through the dependence graph. Thus a program represented as a dependence graph does not violate confidentiality, if and only if

$$\forall a \in dom(R) : \forall\, x \in BS(a) \cap dom(P) : P(x) \leq R(a)$$

That is, the backward slice from a node a with a required security level R(a) must not contain a node x that has a higher security level P(x). Usually, the number of nodes that have a specified security level is low, e.g. points of output. Therefore, the above criterion can easily be transformed into an algorithm that checks a program for confidentiality.

7.2.2 SDG-based Confidentiality Check

To check an information flow, the following property must hold. For every node in the dependence graph that has a required security level specified, compute the backward slice,

and check that no node in the slice has a higher provided security level specified.

Checking each node separately allows a simple yet powerful diagnosis in the case of a security violation: If a node x in the backward slice BS(a) has a provided security level that is too large (P(x) > R(a)), the responsible nodes can be computed by chop(x, a). The chop computes all nodes that are part of the path from node x to node a, thus it contains all nodes that may be involved in the propagation from x's security level to a. If there is no ordering between the provided and the required security levels, the dom(R) and dom(P) should be treated as disjoint sets. Any information flow between these two domains is considered to be information leak.

7.2.3 Example

```
1 package Prog is
2   procedure Q1(A: in Integer; B: in out Integer; C: out Integer);
3      --# derives B from A, B
4      --#           & C from A,B
5
6 end P;
7
8 package body Prog is
9   procedure Q1(A: in Integer; B: in out Integer; C: out Integer) is
10        m : Integer
11    begin
12       m = A + B *2;
13       B = m + A
14       C = A + B;
15    end Q1;
16 end P;
```

Figure 7.2: *Example program to illustrate chopping*

Policy

```
Policy ex1:
  Domain {high, low};
  Order low <= high ;
```

Query

```
Check case1:
  check ex1 with provided {Prog.Q1(in A)} : low, {Prog.Q1(in B)} : high
                required {Prog.Q1(out C)} : high;
```

Using the above policy on the program in the Figure 8.4, we are able to specify that there are two domains *low* and *high* with the ordering that the domains i.e the domain *low* is less secure than the domain *high*. By the above query, we can specify that *Init A* of procedure *Q1* provides information that are classified as *low* and the *Init B* of procedure *Q1* provides information classification as *high*. Also we say that to be a valid information flow, the information received by the node *Final C* of procedure *Q1* should be of level of no greater than *high*.

This is verified using the technique discussed under 7.2.2. A backward slice is computed from the node with required security level, which could be *Final C* in this example. The resultant of the slice contains two nodes *Init A* and *Init B* with provided security level. The provided security level is compared with the required security level based on the order specified in the policy. The provided level of *Init A* and *Init B* are less than or equal to the required level of *Final C*. There for this query results in boolean true and an empty set to mention that the verification of the query is success and there is non path contributing to information leakage.

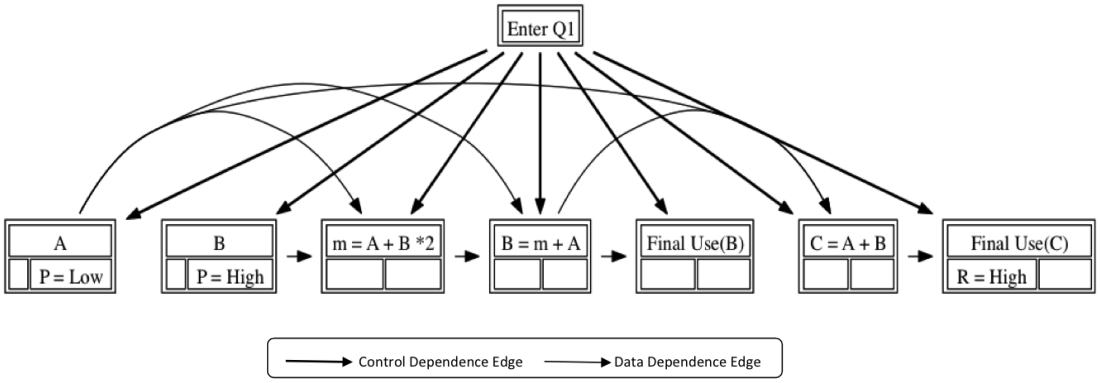

Figure 7.3: *System Dependence graph for the program in 8.4*

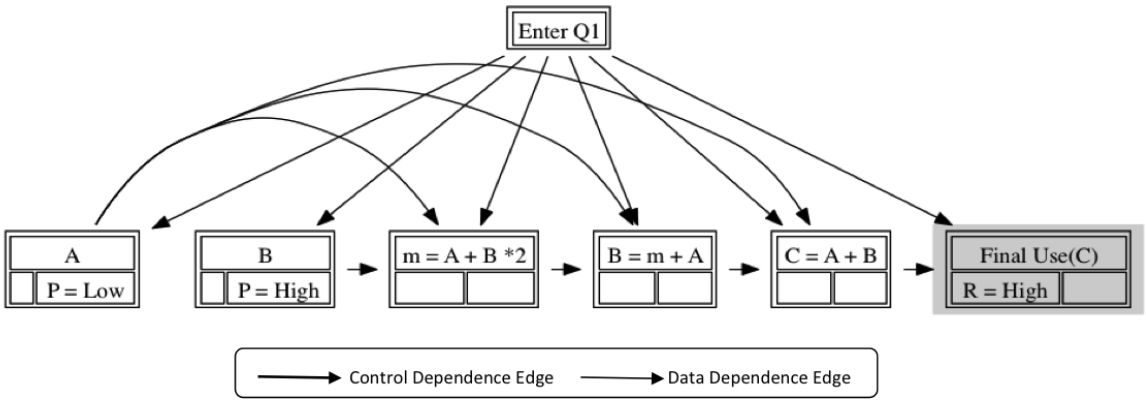

Figure 7.4: *Backward slice on SDG in 7.3*

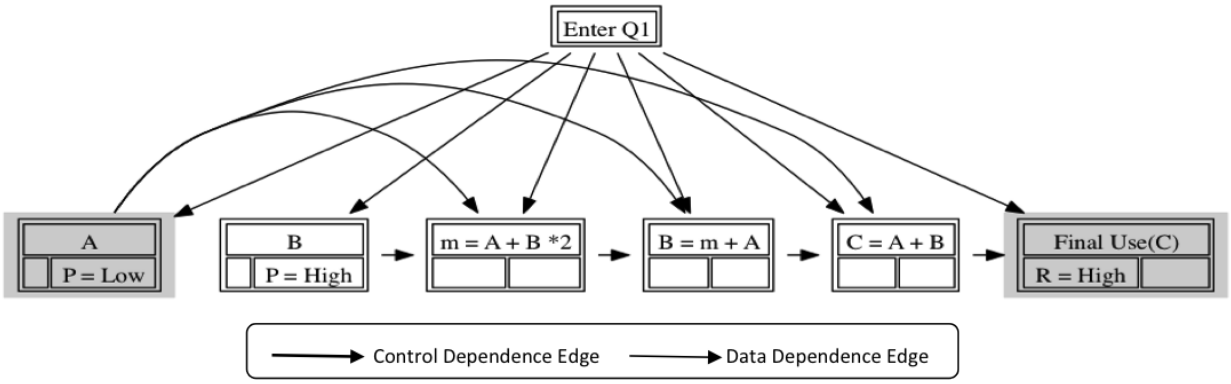

Figure 7.5: *Comparing the first provided and required classification level*

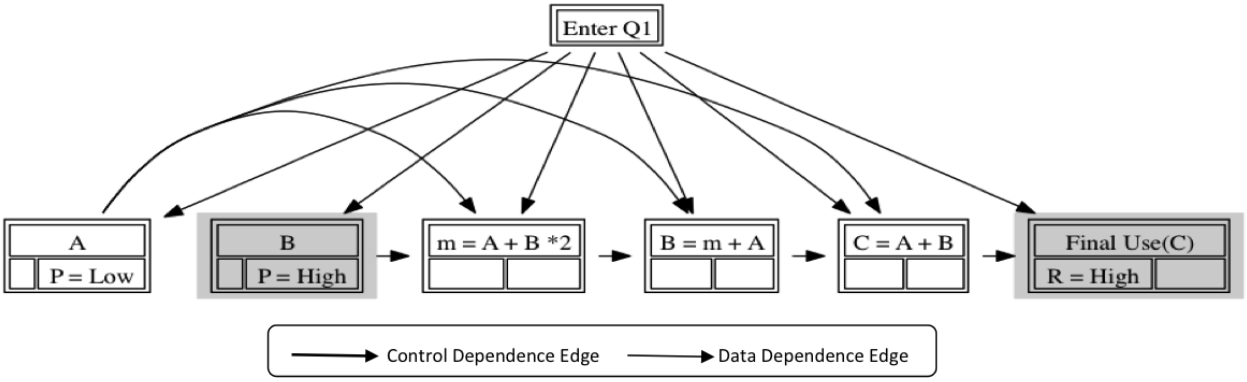

Figure 7.6: *Comparing the provided and required classification level*

Query

```
Check case2:
  check ex1 with provided {Prog.Q1(in A)} : low,
                          {Prog.Q1(in B)} : high
               required {Prog.Q1(out C)} : Low;
```

Verifying the above query to the program in 8.4. The figure 7.7 shows the SDG generated with respect to this query. There are two provided nodes *Init A* and *Init B* with classification as *low* and *high* respectively. There is a required node *Final C* with classification *low*.

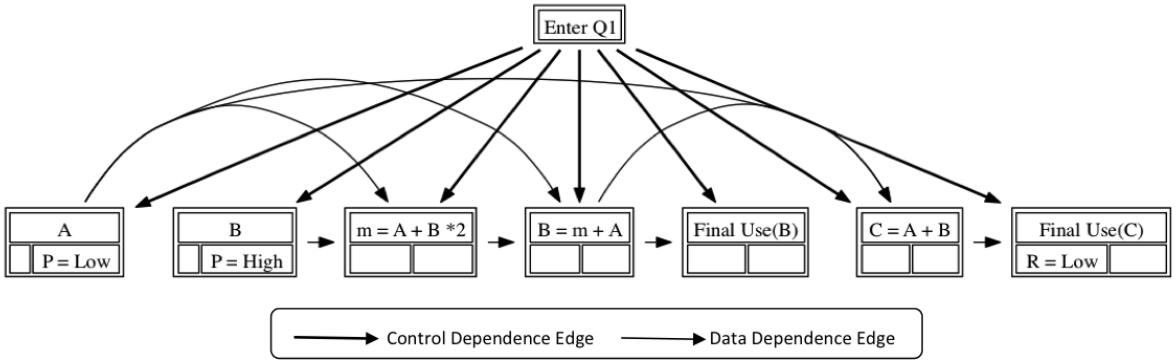

Figure 7.7: *SDG for the program in 7.3*

The backward slice on the SDG in Figure 7.3 with respect to *Final C* is shown in Figure 7.8. There are two nodes with the provided classification. Comparing the first node *Init A* with *Final C*, both the security levels are equal, so it is valid.

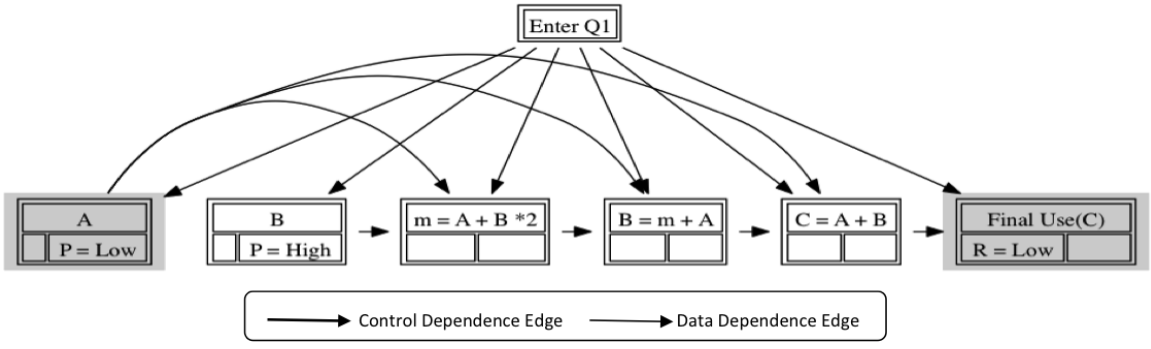

Figure 7.8: *Comparing the provided and required classification level*

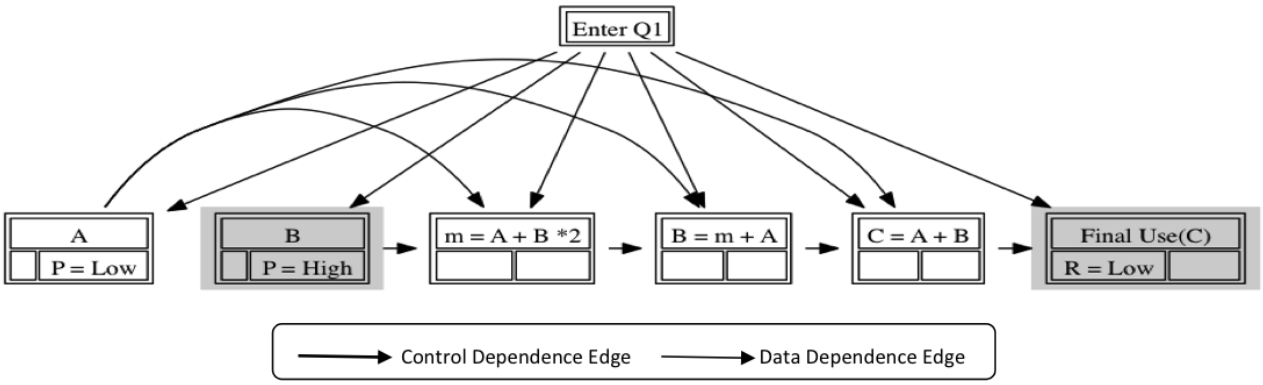

Figure 7.9: *Comparing the provided and required classification level*

When comparing the *Final C* with next node in the provided domain, the provided level is higher than the required node level. This is shown in figure 7.9. The path contributing to the leak is computed by performing a chop with the respective nodes, the Figure 7.10 show the chop from *Final C* and *Init B*.

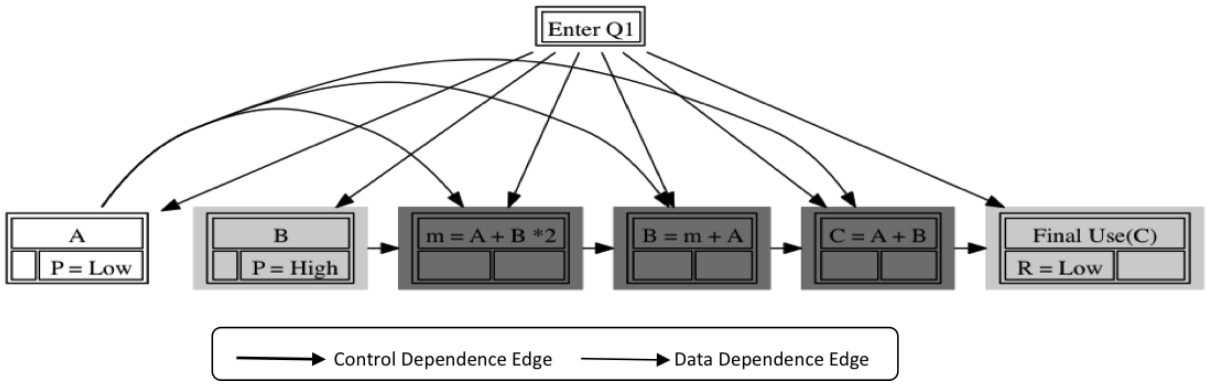

Figure 7.10: *Chop of failure verification*

Chapter 8

MILS - Mailbox

In this chapter we provide a Multiple Independent Levels of Security (MILS) Example to illustrate the information flow control technique discussed in this thesis.

8.1 Mailbox Example

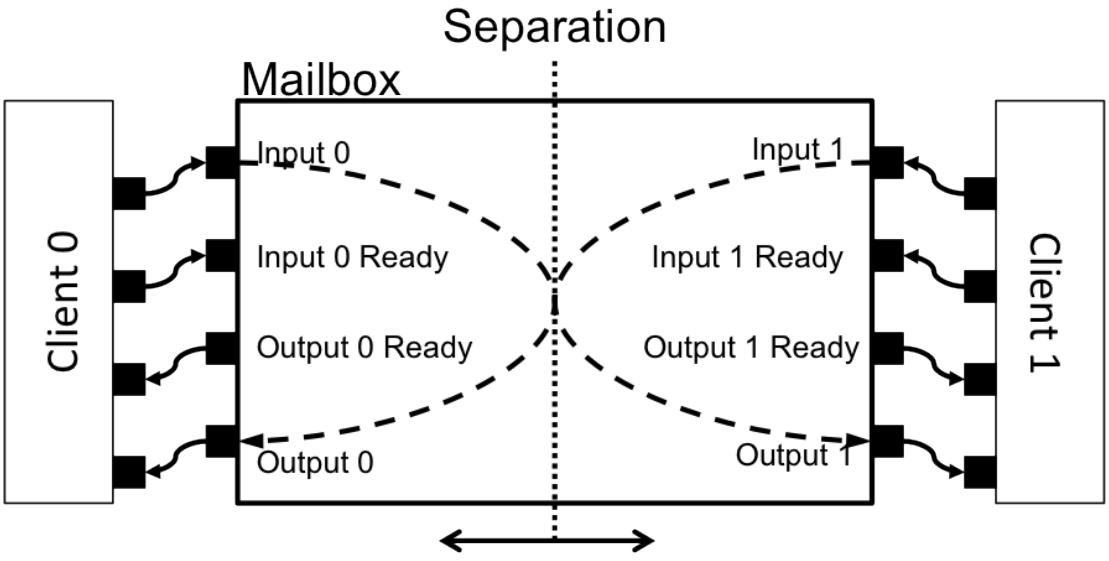

Figure 8.1: *Simple MILS - mailbox*[3]

Figure 8.1 illustrates the MILS component used by Rockwell Collins engineer to demonstrate information flow issues in MILS component. The "Mailbox" component is a mediator

between two clients running in two different partitions of a separation kernel. To send some data from *Client 0* to *Client 1*, *Client 0* writes the data into the memory segment *Input 0*. The *Input 0* is shared between *Client 0* and the mailbox. Then *Client 0* sets the *Input 0 Ready* flag, to inform the mailbox that new input is placed in the *Input 0* memory segment. The mailbox process polls its ready flags to check the data is allowed to transfer from one end to another. *Client 0* can communicate with *Client 1* only when *Input 0 Ready* is set and the *Output 1 Ready* is cleared. Then the mailbox transfers the data from *Input 0* to *Output 1*, clears the *Input 0 Ready* flag and sets the *Output 1 Ready* flag. The *Output 1 Ready* flag is set to inform the *Client 1* that new data is placed in *Output 1*. When *Client 1* consumes the data, it clears the *Output 1 Ready* flag. The communication from *Client 1* to *Client 0* follow a symmetric set of steps.

8.2 Machine Step Procedure Analyses

Figure 8.2 is a fragment of mailbox implementation, full program is provided in appendix **??**, it captures the core process discussed in section 8.1. The derives clause captures the information flow of this procedure. The *Output 1* is derived from *Input 0*, *Input 0 Ready*, *Output 1 Ready* and *Output 1*. Similarly the *Output 0* is derived from *Input 1*, *Output 0 Ready*, *Input 1 Ready* and *Output 0*. We need to verify that there are two separate information channels in the mailbox and there is no interference between them.

We can capture the *Input 0*, *Input 1*, *Output 0* and *Output 1* in the policy language by the queries described in figure 8.3.

The information policy is specified in the figure 8.4, note that there is no order between A and B, which means A and B are disjoint. Therefore should be no information flow between the path A and path B.

```
1   procedure MACHINE_STEP ;
2     --# global in out INTEGER_INPUT_0_READY,
3     --#                 INTEGER_INPUT_1_READY,
4     --#                 INTEGER_OUTPUT_0_READY,
5     --#                 INTEGER_OUTPUT_1_READY,
6     --#                 INTEGER_OUTPUT_0_DATA,
7     --#                 INTEGER_OUTPUT_1_DATA;
8     --#         in      INTEGER_INPUT_0_DATA,
9     --#                 INTEGER_INPUT_1_DATA;
10    --# derives INTEGER_OUTPUT_0_DATA from INTEGER_INPUT_1_DATA,
11    --#                                    INTEGER_OUTPUT_0_READY,
12    --#                                    INTEGER_OUTPUT_0_DATA,
13    --#                                    INTEGER_INPUT_1_READY &
14    --#          INTEGER_OUTPUT_1_DATA from INTEGER_INPUT_0_DATA,
15    --#                                    INTEGER_INPUT_0_READY,
16    --#                                    INTEGER_OUTPUT_1_DATA,
17    --#                                    INTEGER_OUTPUT_1_READY &
18    --#          INTEGER_INPUT_0_READY from INTEGER_INPUT_0_READY,
19    --#                                    INTEGER_OUTPUT_1_READY &
20    --#          INTEGER_INPUT_1_READY from INTEGER_INPUT_1_READY,
21    --#                                    INTEGER_OUTPUT_0_READY &
22    --#          INTEGER_OUTPUT_0_READY from INTEGER_OUTPUT_0_READY,
23    --#                                     INTEGER_INPUT_1_READY &
24    --#          INTEGER_OUTPUT_1_READY from INTEGER_OUTPUT_1_READY,
25    --#                                     INTEGER_INPUT_0_READY;
26
27  procedure Main;
28
29  procedure MACHINE_STEP
30  is
31      DATA_0 : INTEGER;
32      DATA_1 : INTEGER;
33    begin
34      if INPUT_0_READY and OUTPUT_1_CONSUMED then
35        DATA_0 := READ_INPUT_0;
36        NOTIFY_INPUT_0_CONSUMED;
37        WRITE_OUTPUT_1(DATA_0);
38        NOTIFY_OUTPUT_1_READY;
39      end if;
40      if INPUT_1_READY and OUTPUT_0_CONSUMED then
41        DATA_1 := READ_INPUT_1;
42        NOTIFY_INPUT_1_CONSUMED;
43        WRITE_OUTPUT_0(DATA_1);
44        NOTIFY_OUTPUT_0_READY;
45      end if;
46  end MACHINE_STEP;
```

Figure 8.2: *Mailbox code fragment : Machine_Step Procedure*

```
query Input_0:
   {Mailbox.MACHINE_STEP(in Mailbox.INTEGER_INPUT_0_DATA)};

query Input_1:
   {Mailbox.MACHINE_STEP(in Mailbox.INTEGER_INPUT_1_DATA)};

query Output_0:
   {Mailbox.MACHINE_STEP(out Mailbox.INTEGER_OUTPUT_0_DATA)};

query Output_1:
   {Mailbox.MACHINE_STEP(out Mailbox.INTEGER_OUTPUT_1_DATA)};
```

Figure 8.3: *Queries to capture Input and Output*

```
policy path:
   domain {A, B}
```

Figure 8.4: *Security policy*

Output 0 to *Input 1*

The query *pathA* specifies that *Input 1* and *Output 0* belongs to classification *A*. As both are equal, the information flow between them is valid, which is indicated by the result 'true'.

```
check pathA:
   check path with provided Input_1 : A required Output_0 : A;

Result:-
pathA:
   InitDef$Mailbox$@@INTEGER_INPUT_1_DATA ->
   FinalUse$Mailbox$@@INTEGER_OUTPUT_0_DATA : true
```

Figure 8.5: *Verification of Path A*

Output 1 to *Input 0*

The *pathB* is verified similar to *pathA*.

```
check pathB:
   check path with provided Input_0 : B required Output_1 : B;

Result:-
pathA:
   InitDef$Mailbox$@@INTEGER_INPUT_0_DATA ->
   FinalUse$Mailbox$@@INTEGER_OUTPUT_1_DATA : true
```

Figure 8.6: *Verification of Path B*

No Interference

To make sure that there is no interference of data between the *path A* and *path B*, we verify them together to with appropriate security domains.

```
check Both_A_and_B:
  check path with
         provided
                 {Mailbox.MACHINE_STEP(in Mailbox.INTEGER_INPUT_0_DATA)} : B,
                 {Mailbox.MACHINE_STEP(in Mailbox.INTEGER_INPUT_1_DATA)} : A
         required
                 {Mailbox.MACHINE_STEP(out Mailbox.INTEGER_OUTPUT_1_DATA)} : B,
                 {Mailbox.MACHINE_STEP(out Mailbox.INTEGER_OUTPUT_0_DATA)} : A ;
Result:-
Both_A_and_B:
        InitDef$Mailbox$@@INTEGER_INPUT_0_DATA ->
        FinalUse$Mailbox$@@INTEGER_OUTPUT_1_DATA : true
        InitDef$Mailbox$@@INTEGER_INPUT_1_DATA ->
        FinalUse$Mailbox$@@INTEGER_OUTPUT_0_DATA : true
```

Figure 8.7: *Checking for Interference*

Failure Case

Suppose we assume that the *Output 1* belongs to the *path A* but the *Input 0* belongs to *path B*. Then there is a flow between the *Input 0* to *Output 1* which is not valid.

```
check Both_A_and_B_bad:
  check path with
          provided
                  {Mailbox.MACHINE_STEP(in Mailbox.INTEGER_INPUT_0_DATA)} : B,
                  {Mailbox.MACHINE_STEP(in Mailbox.INTEGER_INPUT_1_DATA)} : A
          required
                  {Mailbox.MACHINE_STEP(out Mailbox.INTEGER_OUTPUT_1_DATA)} : A,
                  {Mailbox.MACHINE_STEP(out Mailbox.INTEGER_OUTPUT_0_DATA)} : A ;
Result:-
Both_A_and_B_bad:
        InitDef$Mailbox$@@INTEGER_INPUT_1_DATA ->
        FinalUse$Mailbox$@@INTEGER_OUTPUT_0_DATA : true
        InitDef$Mailbox$@@INTEGER_INPUT_0_DATA ->
        FinalUse$Mailbox$@@INTEGER_OUTPUT_1_DATA : false
        (continued with list of nodes contributing the leak)
```

Figure 8.8: *Checking for Interference - Failure*

Chapter 9

Conclusion

This thesis offers a technique to perform system level information flow analysis for spark language. This is information analysis framework is integrated into an Eclipse plug-in. The Eclipse plug-in offers better presentation and interactive usability.

The policy language is expressive enough to perform pretty complex information flow analysis and also to effectively specify and verify security policies. This tool helps the developer to analyze and verify MILS standard embedded system.

9.1 Future Work

1. Conditional information flow - The present tool captures the information flow property in the system irrespective of the condition implied in the information flow. This looses precession, where conditional information flow is required. This could be remedied by including path condition generators and capturing the conditional aspect of the information flow.

2. The present verification tool supports not the classification of the information but not de-classification. With out the de-classification feature it is impossible to use it in real world as every system needs to be de-classified at user interaction points. This will be considered as the next crucial step of improvement in the policy language and verification tool. This has to be combined with the conditional information flow

analyses to classify and de-classify under certain condition.

3. The policy language need to be improved to accommodate the conditional information flow and declassification. The editor for policy language in Eclipse plug-in is left for future work. The results of the policy language is represented as a tabular view in Eclipse. This view needs to be upgrade to display specified policies and verification results with different color scheme for different information flow channels.

There are more interesting stuff that are possible with the policy language, one of which is the capability to specify higher level policies and refining and verifying the policy at various level until actual implementation is verified against the policy. This feature will be help the developer to stick with specification throughout the engineering process.

Bibliography

[1] Rockwell Collins Secure One, http://www.rockwellcollins.com/sitecore/content/Data/Products/Information_Assurance/Cross_Domain_Solutions/SecureOne_Cross_Domain_Technologies.aspx.

[2] L. Zhebg and S. Sdanewic. A. C. Myers, N. Nystrom. Java information flow. http://www.cornell.edu/jif.

[3] Torben Amtoft, John Hatcliff, Edwin Rodrguez, Robby, Jonathan Hoag, and David Greve. Specification and checking of software contracts for conditional information flow, 2007.

[4] Roderick Chapman and Adrian Hilton. Enforcing security and safety models with an information flow analysis tool. *Proceedings of the 2004 Annual ACM SIGAda International Conference on Ada: The Engineering of Correct and Reliable Software for Real-Time Distributed Systems using Ada and Related Technologies 2004*, pages 39–46, 2004.

[5] M. P. Heimdahl and C. L. Heitmeyer. Formal methods for developing high assurance computer systems. *Working group report. in Proceedings, Second IEEE Workshop on Industrial-Strength Formal Techniques (WIFT'98)*, 1998.

[6] Susan Horwitz Thomas Reps and David Binkley. Interprocedural slicing using dependence graphs. *In Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation*, pages 35–46, 1988.

[7] Thomas Reps and Genevieve Rosay. Precise interprocedural chopping. *In SIGSOFT '95: Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 41–52, 1995.

[8] John Hatcliff Matthew B. Dwyer and Hongjun Zheng. Slicing software for model construction. *Journal of Higher-order and Symbolic Computation*, 13(4):315–353, 2000.

[9] J. Des Rivières and J. Wiegand. Eclipse: a platform for integrating development tools. *IBM Systems Journal*, pages 371–383, 2004.

[10] Barnes. J. *High Integrity Software: The SPARK Approach to Safety and Security*, volume 1st Ed. Addison-Wesley Longman Publishing Co. Inc. ISBN 0-321-13616-0, 2003.

[11] Andrew C. Myers. Jflow: Practical mostly-static information flow control. *In Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, 1999.

[12] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*, volume 2nd Ed. Springer-Verlag, 2005.

[13] John Rushby. Separation and integration in mils. *The MILS Constitution*, 2008.

Appendix A

Mailbox Program

The complete Mailbox program is provided in this section.

```
1  -- This is a SPARK-Ada version of the simple C mailbox example
2  -- provided by Rockwell Collins. The example is enriched with
3  -- SPARK annotations. The purpose of this simple program is to
4  -- transmit data from one entity to another through a mediator
5  -- (the mailbox).
6  -- @author Edwin Rodrgues
7
8  package Mailbox
9  --# own INTEGER_INPUT_0_READY, INTEGER_INPUT_0_DATA,
10 --#     INTEGER_OUTPUT_0_READY, INTEGER_OUTPUT_0_DATA,
11 --#     INTEGER_INPUT_1_READY, INTEGER_INPUT_1_DATA,
12 --#     INTEGER_OUTPUT_1_READY, INTEGER_OUTPUT_1_DATA;
13 --# initializes INTEGER_INPUT_0_READY, INTEGER_INPUT_0_DATA,
14 --#     INTEGER_OUTPUT_0_READY, INTEGER_OUTPUT_0_DATA,
15 --#     INTEGER_INPUT_1_READY, INTEGER_INPUT_1_DATA,
16 --#     INTEGER_OUTPUT_1_READY, INTEGER_OUTPUT_1_DATA;
17 is
18
19   INTEGER_INPUT_0_READY : BOOLEAN := TRUE;
20   INTEGER_INPUT_0_DATA : INTEGER := -1;
21   INTEGER_OUTPUT_0_READY : BOOLEAN := TRUE;
22   INTEGER_OUTPUT_0_DATA : INTEGER := -1;
23
24   INTEGER_INPUT_1_READY : BOOLEAN := TRUE;
25   INTEGER_INPUT_1_DATA : INTEGER := -1;
26   INTEGER_OUTPUT_1_READY : BOOLEAN := TRUE;
27   INTEGER_OUTPUT_1_DATA : INTEGER := -1;
28
29   function INPUT_0_CONSUMED return BOOLEAN;
30   --# global in INTEGER_INPUT_0_READY;
```

Figure A.1: *MILS - Mailbox Program*

```
31  function INPUT_0_READY return BOOLEAN;
32  --# global in INTEGER_INPUT_0_READY;
33
34  function OUTPUT_0_CONSUMED return BOOLEAN;
35  --# global in INTEGER_OUTPUT_0_READY;
36
37  function OUTPUT_0_READY return BOOLEAN;
38  --# global in INTEGER_OUTPUT_0_READY;
39
40  function INPUT_1_CONSUMED return BOOLEAN;
41  --# global in INTEGER_INPUT_1_READY;
42
43  function INPUT_1_READY return BOOLEAN;
44  --# global in INTEGER_INPUT_1_READY;
45
46  function OUTPUT_1_CONSUMED return BOOLEAN;
47  --# global in INTEGER_OUTPUT_1_READY;
48
49  function OUTPUT_1_READY return BOOLEAN;
50  --# global in INTEGER_OUTPUT_1_READY;
51
52  procedure NOTIFY_INPUT_0_CONSUMED;
53  --# global out INTEGER_INPUT_0_READY;
54  --# derives INTEGER_INPUT_0_READY from ;
55
56  procedure NOTIFY_INPUT_0_READY;
57  --# global out INTEGER_INPUT_0_READY;
58  --# derives INTEGER_INPUT_0_READY from ;
59
60  procedure NOTIFY_OUTPUT_0_CONSUMED;
61  --# global out INTEGER_OUTPUT_0_READY;
62  --# derives INTEGER_OUTPUT_0_READY from ;
63
64  procedure NOTIFY_OUTPUT_0_READY;
65  --# global out INTEGER_OUTPUT_0_READY;
66  --# derives INTEGER_OUTPUT_0_READY from ;
67
68  procedure NOTIFY_INPUT_1_CONSUMED;
69  --# global out INTEGER_INPUT_1_READY;
70  --# derives INTEGER_INPUT_1_READY from ;
71
72  procedure NOTIFY_INPUT_1_READY;
73  --# global out INTEGER_INPUT_1_READY;
74  --# derives INTEGER_INPUT_1_READY from ;
75
76  procedure NOTIFY_OUTPUT_1_CONSUMED;
77  --# global out INTEGER_OUTPUT_1_READY;
78  --# derives INTEGER_OUTPUT_1_READY from ;
```

Figure A.2: *MILS - Mailbox Program continued*

```
79   procedure NOTIFY_OUTPUT_1_READY;
80   --# global out INTEGER_OUTPUT_1_READY;
81   --# derives INTEGER_OUTPUT_1_READY from ;
82
83   function READ_INPUT_0 return INTEGER;
84   --# global in INTEGER_INPUT_0_DATA;
85
86   function READ_OUTPUT_0 return INTEGER;
87   --# global in INTEGER_OUTPUT_0_DATA;
88
89   function READ_INPUT_1 return INTEGER;
90   --# global in INTEGER_INPUT_1_DATA;
91
92   function READ_OUTPUT_1 return INTEGER;
93   --# global in INTEGER_OUTPUT_1_DATA;
94
95   procedure WRITE_INPUT_0(Data : in INTEGER);
96   --# global out INTEGER_INPUT_0_DATA;
97   --# derives INTEGER_INPUT_0_DATA from Data;
98
99   procedure WRITE_OUTPUT_0(Data : in INTEGER);
100  --# global out INTEGER_OUTPUT_0_DATA;
101  --# derives INTEGER_OUTPUT_0_DATA from Data;
102
103  procedure WRITE_INPUT_1(Data : in INTEGER);
104  --# global out INTEGER_INPUT_1_DATA;
105  --# derives INTEGER_INPUT_1_DATA from Data;
106
107  procedure WRITE_OUTPUT_1(Data : in INTEGER);
108  --# global out INTEGER_OUTPUT_1_DATA;
109  --# derives INTEGER_OUTPUT_1_DATA from Data;
```

Figure A.3: *MILS - Mailbox Program continued*

```
110   procedure MACHINE_STEP ;
111     --# global in out INTEGER_INPUT_0_READY, INTEGER_INPUT_1_READY,
112     --#                 INTEGER_OUTPUT_0_READY, INTEGER_OUTPUT_1_READY,
113     --#                 INTEGER_OUTPUT_0_DATA, INTEGER_OUTPUT_1_DATA;
114     --#         in      INTEGER_INPUT_0_DATA, INTEGER_INPUT_1_DATA;
115     --# derives INTEGER_OUTPUT_0_DATA from INTEGER_INPUT_1_DATA,
116     --#                                                 INTEGER_OUTPUT_0_READY,
117     --#                                                 INTEGER_OUTPUT_0_DATA,
118     --#                                                 INTEGER_INPUT_1_READY &
119     --#         INTEGER_OUTPUT_1_DATA from INTEGER_INPUT_0_DATA,
120     --#                                                 INTEGER_INPUT_0_READY,
121     --#                                                 INTEGER_OUTPUT_1_DATA,
122     --#                                                 INTEGER_OUTPUT_1_READY &
123     --#         INTEGER_INPUT_0_READY from INTEGER_INPUT_0_READY,
124     --#                                                 INTEGER_OUTPUT_1_READY &
125     --#         INTEGER_INPUT_1_READY from INTEGER_INPUT_1_READY,
126     --#                                                 INTEGER_OUTPUT_0_READY &
127     --#         INTEGER_OUTPUT_0_READY from INTEGER_OUTPUT_0_READY,
128     --#                                                  INTEGER_INPUT_1_READY &
129     --#         INTEGER_OUTPUT_1_READY from INTEGER_OUTPUT_1_READY,
130     --#                                                  INTEGER_INPUT_0_READY;
131 procedure Main;
132   --# global in out INTEGER_INPUT_0_READY, INTEGER_INPUT_1_READY,
133   --#                INTEGER_OUTPUT_0_READY, INTEGER_OUTPUT_1_READY,
134   --#                INTEGER_OUTPUT_0_DATA, INTEGER_OUTPUT_1_DATA;
135   --#        in      INTEGER_INPUT_0_DATA, INTEGER_INPUT_1_DATA;
136   --# derives INTEGER_OUTPUT_0_DATA from INTEGER_INPUT_1_DATA,
137   --#                                                 INTEGER_OUTPUT_0_READY,
138   --#                                                 INTEGER_OUTPUT_0_DATA,
139   --#                                                 INTEGER_INPUT_1_READY &
140   --#         INTEGER_OUTPUT_1_DATA from INTEGER_INPUT_0_DATA,
141   --#                                                 INTEGER_INPUT_0_READY,
142   --#                                                 INTEGER_OUTPUT_1_DATA,
143   --#                                                 INTEGER_OUTPUT_1_READY &
144   --#         INTEGER_INPUT_0_READY from INTEGER_INPUT_0_READY,
145   --#                                                 INTEGER_OUTPUT_1_READY &
146   --#         INTEGER_INPUT_1_READY from INTEGER_INPUT_1_READY,
147   --#                                                 INTEGER_OUTPUT_0_READY &
148   --#         INTEGER_OUTPUT_0_READY from INTEGER_OUTPUT_0_READY,
149   --#                                                  INTEGER_INPUT_1_READY &
150   --#         INTEGER_OUTPUT_1_READY from INTEGER_OUTPUT_1_READY,
151   --#                                                  INTEGER_INPUT_0_READY;
152 end Mailbox;
```

Figure A.4: *MILS - Mailbox Program continued*

```
153 package body Mailbox
154 is
155   function INPUT_0_CONSUMED return BOOLEAN is
156   begin
157     return not INTEGER_INPUT_0_READY;
158   end INPUT_0_CONSUMED;
159
160   function INPUT_0_READY return BOOLEAN is
161   begin
162     return INTEGER_INPUT_0_READY;
163   end INPUT_0_READY;
164
165   function OUTPUT_0_CONSUMED return BOOLEAN is
166   begin
167     return not INTEGER_OUTPUT_0_READY;
168   end OUTPUT_0_CONSUMED;
169
170   function OUTPUT_0_READY return BOOLEAN is
171   begin
172     return INTEGER_OUTPUT_0_READY;
173   end OUTPUT_0_READY;
174   function INPUT_1_CONSUMED return BOOLEAN is
175   begin
176     return not INTEGER_INPUT_1_READY;
177   end INPUT_1_CONSUMED;
178
179   function INPUT_1_READY return BOOLEAN is
180   begin
181     return INTEGER_INPUT_1_READY;
182   end INPUT_1_READY;
183
184   function OUTPUT_1_CONSUMED return BOOLEAN is
185   begin
186     return not INTEGER_OUTPUT_1_READY;
187   end OUTPUT_1_CONSUMED;
188
189   function OUTPUT_1_READY return BOOLEAN is
190   begin
191     return INTEGER_OUTPUT_1_READY;
192   end OUTPUT_1_READY;
193
194   procedure NOTIFY_INPUT_0_CONSUMED is
195   begin
196     INTEGER_INPUT_0_READY := FALSE;
197   end NOTIFY_INPUT_0_CONSUMED;
198
199   procedure NOTIFY_INPUT_0_READY is
200   begin
201     INTEGER_INPUT_0_READY := TRUE;
202   end NOTIFY_INPUT_0_READY;
```

Figure A.5: *MILS - Mailbox Program continued*

```
203   procedure NOTIFY_OUTPUT_0_CONSUMED is
204   begin
205     INTEGER_OUTPUT_0_READY := FALSE;
206   end NOTIFY_OUTPUT_0_CONSUMED;
207
208   procedure NOTIFY_OUTPUT_0_READY is
209   begin
210     INTEGER_OUTPUT_0_READY := TRUE;
211   end NOTIFY_OUTPUT_0_READY;
212
213   procedure NOTIFY_INPUT_1_CONSUMED is
214   begin
215     INTEGER_INPUT_1_READY := FALSE;
216   end NOTIFY_INPUT_1_CONSUMED;
217
218   procedure NOTIFY_INPUT_1_READY is
219   begin
220     INTEGER_INPUT_1_READY := TRUE;
221   end NOTIFY_INPUT_1_READY;
222   procedure NOTIFY_OUTPUT_1_CONSUMED is
223   begin
224     INTEGER_OUTPUT_1_READY := FALSE;
225   end NOTIFY_OUTPUT_1_CONSUMED;
226
227   procedure NOTIFY_OUTPUT_1_READY is
228   begin
229     INTEGER_OUTPUT_1_READY := TRUE;
230   end NOTIFY_OUTPUT_1_READY;
231
232   function READ_INPUT_0 return INTEGER is
233   begin
234   return INTEGER_INPUT_0_DATA;
235   end READ_INPUT_0;
236
237   function READ_OUTPUT_0 return INTEGER is
238   begin
239     return INTEGER_OUTPUT_0_DATA;
240   end READ_OUTPUT_0;
241
242   function READ_INPUT_1 return INTEGER is
243   begin
244     return INTEGER_INPUT_1_DATA;
245   end READ_INPUT_1;
246
247   function READ_OUTPUT_1 return INTEGER is
248   begin
249     <<tag>>return INTEGER_OUTPUT_1_DATA;
250   end READ_OUTPUT_1;
```

Figure A.6: *MILS - Mailbox Program continued*

```
251    procedure WRITE_INPUT_0(Data : in INTEGER) is
252    begin
253      INTEGER_INPUT_0_DATA := Data;
254    end WRITE_INPUT_0;
255
256    procedure WRITE_OUTPUT_0(Data : in INTEGER) is
257    begin
258      <<test>>INTEGER_OUTPUT_0_DATA := Data;
259    end WRITE_OUTPUT_0;
260
261    procedure WRITE_INPUT_1(Data : in INTEGER) is
262    begin
263      INTEGER_INPUT_1_DATA := Data;
264    end WRITE_INPUT_1;
265
266    procedure WRITE_OUTPUT_1(Data : in INTEGER) is
267    begin
268      INTEGER_OUTPUT_1_DATA := Data;
269    end WRITE_OUTPUT_1;
270  procedure MACHINE_STEP
271    is
272        DATA_0 : INTEGER;
273        DATA_1 : INTEGER;
274      begin
275        if INPUT_0_READY and OUTPUT_1_CONSUMED then
276          DATA_0 := READ_INPUT_0;
277          NOTIFY_INPUT_0_CONSUMED;
278          WRITE_OUTPUT_1(DATA_0);
279          NOTIFY_OUTPUT_1_READY;
280        end if;
281        if INPUT_1_READY and OUTPUT_0_CONSUMED then
282          DATA_1 := READ_INPUT_1;
283          NOTIFY_INPUT_1_CONSUMED;
284          WRITE_OUTPUT_0(DATA_1);
285          NOTIFY_OUTPUT_0_READY;
286        end if;
287    end MACHINE_STEP;
288
289    procedure Main
290    is
291    begin
292     for J in Integer range 1..10 loop
293       MACHINE_STEP;
294     end loop ;
295    end Main;
296  end Mailbox;
```

Figure A.7: *MILS - Mailbox Program continued*

Appendix B

Data From SDG Construction Experiment

The collected data are from the regression test suit built to test the SDG.

Program	**Average Time in Seconds**	**Nodes in SDG**	**Edges in SDG**
MILS - MMR	5.32	1502	2202
MILS - Guard	40.14	722	1026
Mailbox	3.01	526	466
Simple call three	1.56	172	152
ArraySet	1.95	85	106
Inter-Package dependency	1.56	77	68
Router-Guard	1.5	129	118
Alir Dependence Test 1	2.16	410	319
Alir Dependence Test 2	3.94	1520	1635
Simple Function	1.15	89	68
Package Demo	1.45	42	31
Stack-praxis	1.10	62	49
Refinement Test	1.17	72	59
Array Two Dimensional	1.5	76	64

Table B.1: *Experimental Data for construction of SDG*

The MILS- Guard example consumes more time than it is expected because, of multi dimensional array and various inter package information flows. Around 70% of the execution time is spent on computing the reaching definitions.