Dynamic Analysis of Ada Programs for Comprehension and Quality Measurement

Elaheh Safari Sharifabadi

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montreal, Quebec, Canada
October 2008

Your file Votre référence
ISBN: 978-0-494-45527-2
Our file Notre référence
ISBN: 978-0-494-45527-2

Abstract

Dynamic Analysis of Ada Programs for Comprehension and Quality Measurement

Elaheh Safari Sharifabadi

During maintenance and particularly during corrective and perfective tasks, systems tend to exhibit a weight gain. As a result, their quality tends to degrade. Software comprehension is vital in order to assess system quality. In this research, we aim at deploying dynamic analysis of Ada programs for obtaining comprehension, and applying measurements to assess their quality. Program instrumentation is performed non-intrusively by AspectAda, an aspect-oriented extension to Ada. Events which are required for this analysis are captured as execution traces. We have defined a relational database schema to save execution traces, and a set of queries to obtain measures of quality metrics. New Ada-specific metrics are introduced and existing metrics have been adopted from the literature. Automation is also provided as a proof of concept through a prototypical tool which provides information on the run-time behavior of the system, performs measurements and provides visualization of the run-time behavior of the system through a call graph. An open source Ada program is used as a case study to demonstrate our approach.

Acknowledgments

I would like to express my special gratitude to my supervisor, Dr. Constantinos Constantinides, whose support and guidance contributed to my graduate work and experience. I would like to acknowledge Dr. Olga Ormandjieva for her valuable comments on this dissertation. A sincere thank to all members of Software Maintenance and Evolution Research Group for providing valuable comments on this work.

I owe my loving thanks to my husband, Vahid Safar Nourollah, for his support, encouragement and understanding. I warmly thank my family, who encouraged me during my studies.

This work is partially funded by the Natural Sciences and Engineering Research Council of Canada (NSERC).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many Ada systems written in the 80s and 90s continue to provide value to their stakeholders. However, due to different types of maintenance (particularly perfective and corrective) these systems tend to have gained weight, exhibiting the difficulties associated with software aging and falling into a *legacy* state, providing a dilemma to its stakeholders: On one hand, the system is difficult to understand and maintain since quality and design have been degraded. On the other hand, the system cannot be easily replaced by existing applications, since it fully satisfies the needs of the business rules in the organization [49].

Maintainers of these systems need to first obtain an understanding of the system in order to be able to detect components with degraded quality. System documentation, if at all present, cannot provide much help since it is often incomplete or out of date. The traditional way to understand the system is to manually explore the source code (static analysis). For large-scale programs, however, this process can be tedious and time consuming. An alternative approach is to capture and analyze

the dynamic behavior of the system during the execution time. This way, having a specific need such as evaluating the quality of the system, maintainers can focus and extract a well-defined set of information. One system characteristic which gets directly degraded by software aging is *modularity*. Modularity can be evaluated by exploring the dependencies which are categorized in two sets: First, the relatedness of module elements, and second, the module dependencies, known by the terms *cohesion* and *coupling* respectively.

1.1 Objective and goals of this dissertation

In this research, we discuss an approach to extract information about the dynamic dependencies within and between system packages. Using this information, we also define a quantitative approach to assess coupling, cohesion and modularity. We believe that this approach provides an environment in which maintainers can investigate Ada legacy systems and observe packages whose modularity has been degraded because of software aging.

1.2 Organization of the dissertation

The rest of this dissertation is organized as follows: In Chapter 2, we provide some necessary background on system quality, comprehension and measurement. The motivation behind this research is listed in Chapter 3 and our proposal is outlined in Chapter 4. The methodology to apply the proposal is discussed in Chapter 5 which

provides the way to obtain the program comprehension and subsequently Chapter 6 which discusses the metrics to measure quality factors of the system. In Chapter 7, we validate our approach through applying our proposal on a case study. Automation and tool support are discussed in Chapter 8. In Chapter 9, we discuss related work and finally, we list conclusions and recommendations for further work in Chapter 10.

Chapter 2

Theoretical background

In this chapter, we provide some necessary theoretical background on quality of the system, program comprehension and measurement.

2.1 System quality

ISO 8402[35] defines quality as: "the totality of features and characteristics of a product or a service that bear on its ability to satisfy stated or implied needs." ISO 9126 [36] defines a model to categorize different types of quality characteristics and attributes. As shown in Figure 1, the model is based on three levels: The set of system characteristics is being refined into sets of sub-characteristics, each of which is measured by a set of metrics. ISO 9126 further defines a set of pure internal metrics to measure certain attributes of software design and implementation of the software product that will influence the same or all of the overall software characteristics and sub-characteristics. For maintainability, modularity seems to be the most important

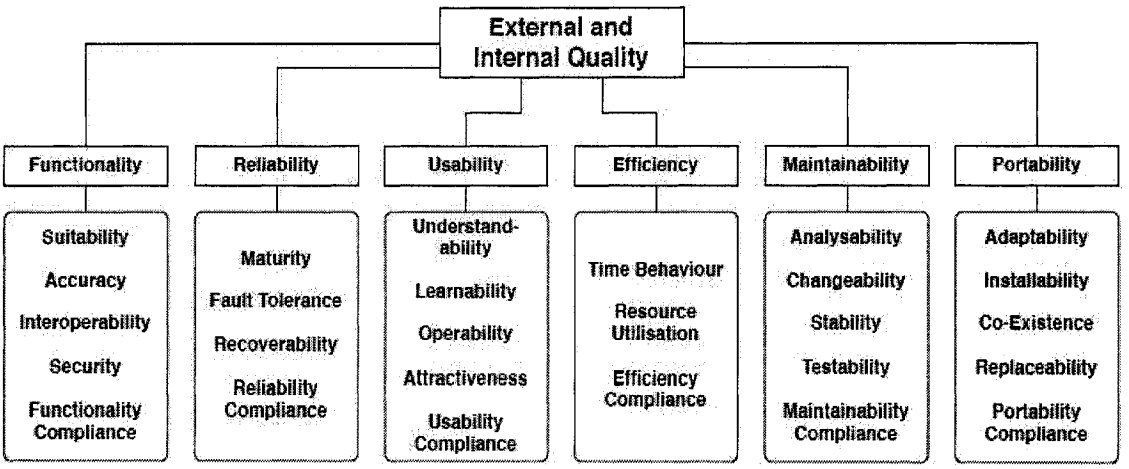

Figure 1: ISO 9126 Quality Model.

internal metric as it affects a number of sub-characteristics such as analysability, changeability, portability and adaptability.

2.2 Measurement

Fenton and Pfleeger in [21] have defined Measurement as "the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to characterize them according to clearly defined rules." In the context of software engineering, measurement can aid in understanding of the system during development and maintenance, control the activities in the system development and improve the processes and product. The measurable entities can be classified into three classes as processes, products and resources. Also the attributes of each entity can be categorized as internal or external attributes. Figure 2 categorizes measurable attributes of product, process and resource entities.

The popularity of the object-oriented paradigm has raised the need for defining

ENTITIES	ATTRIBUTES	
	Internal	*External*
Products		
Specifications	size, reuse, modularity, redundancy, functionality, syntactic correctness, ...	comprehensibility, maintainability, ...
Designs	size, reuse, modularity, coupling, cohesiveness, inheritance, functionality, ...	quality, complexity, maintainability, ...
Code	size, reuse, modularity, coupling, functionality, algorithmic complexity, control-flow structuredness, ...	reliability, usability, maintainability, reusability
Test data	size, coverage level, ...	quality, reusability, ...
...	...	...
Processes		
Constructing specification	time, effort, number of requirements changes, ...	quality, cost, stability, ...
Detailed design	time, effort, number of specification faults found, ...	cost, cost-effectiveness, ...
Testing	time, effort, number of coding faults found, ...	cost, cost-effectiveness, stability, ...
...	...	...
Resources		
Personnel	age, price, ...	productivity, experience, intelligence, ...
Teams	size, communication level, structuredness, ...	productivity, quality, ...
Organisations	size, ISO Certification, CMM level	Maturity, profitability, ...
Software	price, size, ...	usability, reliability, ...
Hardware	price, speed, memory size, ...	reliability, ...
Offices	size, temperature, light, ...	comfort, quality, ...
...	...	...

Figure 2: Classification of software measurement activities [20]

new metrics for measuring the attributes of object-oriented artifacts. Object-oriented measurement is a 20 years old field of research, starting from cost and effort prediction measures. Following that, new internal and external attributes for object-oriented design get introduced such as reliability, complexity, reusability, maintainability, coupling and cohesion. Several measures have been proposed by researchers in this field, while each measure focuses on a specific attribute of the system.

2.3 Modularity

Modularity is defined as "the functional independence of program components" [54]. The definition implies that modularity is a varying property. As such we can say that a high degree of modularity of a component implies a high degree of representation of a single concern and a low degree of dependency on other components. Components with a higher level of modularity tend to be more understandable, reusable and easier to change and evolve during maintenance. Low modularity can be an indication of bad design or ignorant surgery [49]. It occurs when the implementation of a feature or functionality is not well-localized in the decomposition hierarchy of the system. One possible way to evaluate the modularity of a system is to look at the degree to which the activities within a single module are related to one another, introducing the notion of cohesion. Yet another way to determine modularity is to explore the degree to which the modules are cleanly separated from one another, introducing the notion of coupling. Investigating both metrics can give us a better idea [17, 46, 54].

Cohesion is a measure of how strongly the responsibilities of a software module are interrelated [54, 60]. A high degree of cohesion is desirable to avoid having more than one concern being implemented in a single modular unit.

Coupling is the degree to which each program module relies on other modules [54]. As a basic principle of object-oriented programming, there should be a baseline coupling for necessary collaborations between system components [4]. A low degree of coupling is desired as it can contribute to reuse and adaptability, resulting in less costly maintenance.

2.4 Aspect-oriented programming and AspectAda

Despite the success of object-orientation in the effort to achieve separation of concerns, certain properties in object-oriented systems cannot be directly mapped in a one-to-one correspondence from the problem domain to the solution space, and thus they cannot be localized in single modular units. Their implementation ends up cutting across decomposition of the system, resulting in a decreased level of modularity. Crosscutting concerns (or "aspects") include persistence, authentication, synchronization and contract checking. Aspect-Oriented Programming (AOP) [38] explicitly addresses those concerns by introducing the notion of aspect, which is a modular unit of decomposition. Currently there exist many approaches and technologies to support AOP. One notable technology is AspectJ [37], an aspect-oriented extension to the Java language. AspectJ has influenced the design dimensions of several general-purpose aspect-oriented languages. In the AspectJ model, an aspect definition is a new unit of modularity providing behavior to be inserted over functional components. This behavior is defined in procedure-like blocks called *advice* blocks which can be invoked before, after or instead-of (around) some core functionality. However, unlike a procedure, an advice is never explicitly called. Instead, it is only implicitly invoked by an associated construct called a *pointcut* expression. A pointcut expression is a predicate over well-defined points in the execution of the program which are referred to as *join points.*

In earlier work we presented AspectAda [52], an aspect-oriented extension to the

Ada[1] language. AspectAda is an AspectJ-like language, providing a join point model, a pointcut description language and the three different types of advice. The join point model includes calling and executing of subprograms, controlled type initialization and finalization.

2.5 Program comprehension

Program comprehension is the process of obtaining knowledge about a program [58, 61]. It has shown to consume a significantly large proportion of resources during the overall maintenance phase [58], particularly when maintainers are not the initial developers of the system. Furthermore, design artifacts, if at all present, cannot provide complete comprehension since they are often incorrect or incomplete. Comprehension methods rely on analysis of the dependencies between program (or software) elements. A lot of effort has been spent on providing approaches to support and facilitate program comprehension and, as a result, a variety of models of human program comprehension process have been proposed. One extensive survey is provided by von Meyrhauser and Vans in [61]. Following the proposal of comprehension approaches, many tools have been developed to provide comprehension. These tools can be categorized according to the techniques they deploy [65]:

- **Deduction** refers to reasoning from source code to concrete runs. Using this technique, the system analyzer tries to predict what should and should not happen during program execution. For obtaining knowledge concrete runs are

[1] AspectAda initially supported Ada 95 and it was later extended to support Ada 2005.

not required, i.e. there is no need to execute the program to obtain the program runs.

- **Observation** refers to inspecting different program properties in a concrete run. One should first obtain an execution trace of the program using a tracer or debugger, and then inspect program properties or compare them with expected knowledge obtained through deduction.

- **Induction** refers to reasoning from specific facts to general principles. One tries to summarize several observations from program concrete runs to an abstract conclusion. Induction is deployed by visualization tools that gather and summarize dynamic traces like call graph visualization tools.

- **Experimentation** is an advanced technique that determines specific cause(s) for an observed effect. One tries to isolate the real cause by applying several experiences to prove that whenever the cause occurs, the effect will occur and whenever the effect does not occur, it is because the cause has not occurred.

Most of the existing analysis tools are deploying deduction and observation, while integrating experimentation with induction or deduction is currently a main challenge in program comprehension.

2.6 Static and dynamic analysis

Program analysis is the process of analyzing the structure and behavior of computer programs to determine certain properties. There exist two main types of program

analysis: *static analysis* and *dynamic analysis*, which have complementary strengths and weaknesses [18].

Static program analysis is a process to evaluate a system by its structure and it mostly focuses on source code (it may also be applied to object code). Its advantage is that it is *complete*, *sound* and *conservative*. *Completeness* implies that the analysis predicts behavior resulting from all possible scenarios (since it is performed without actually executing the program under consideration). *Soundness* implies that the analysis guarantees that its results are an accurate description of the program behavior, regardless of any actual input. *Conservatism* implies that the analysis reports properties that are guaranteed to be sound but may be weaker (less precise) than desirable. The disadvantage of static analysis is that its results cannot be used for getting precise information. Static analysis tends to produce large amounts of data which can be difficult to understand, summarize, and difficult to extract useful information from. Also, it provides potential paths instead of guaranteed execution paths [41]. In addition, polymorphism and dynamic binding, supported by object-oriented languages, can affect the system behavior only at runtime, thus static analysis cannot fully capture the real behavior and dependencies of the system. In such cases, there is a need for dynamic analysis techniques.

Dynamic program analysis operates by executing a program and examining the execution of events in order to evaluate system properties and behavior. An event refers to any change of program state or behavior and the level of granularity in monitoring events is based on the requirements of analysis. Events are called *execution traces* and they are captured by a process called *instrumentation*. The concept of

dynamic analysis is explored by a number of approaches in the literature [24, 28, 40, 51, 62], each one deploying a different technique to extract and visualize execution traces. An advantage of dynamic analysis is that it is precise because it examines the actual and exact run-time behavior obtained during program execution. It is also efficient as it can be as fast as program execution. A disadvantage of dynamic analysis is that the results may not generalize for future executions, unless input is carefully considered.

2.6.1 Trace extraction techniques for dynamic analysis

There are various techniques for extracting dynamic information at runtime, each one accompanied by a set of advantages and disadvantages, as follows:

- Debugger: Breakpoints can be set at different locations and the information will be shown upon reaching each breakpoint. The advantage of this approach is that the source code will not be modified, but it can considerably reduce the performance. One such example for Ada is GDB [25].

- Source code instrumentation: Probes (commonly print statements) are inserted in different locations in the source code. The advantage of this approach is that it is fast. The disadvantage is that the source code will be modified.

- Binary instrumentation: This technique adds instrumentation to a compiled binary. An advantage of this approach is that it is not intrusive. A difficulty associated with this techniques from the maintainers point of view is that advanced knowledge of binary code is needed to apply this kind of instrumentation.

- Runtime environment instrumentation: Hooks are inserted in the runtime environment by related libraries, so that events of interest will be captured. An advantage of this approach is that it is not intrusive. A disadvantage is the overhead for loading the libraries.

- Compiler assisted: The compiler is associated with the execution of some libraries that instrument the code during compilation. In this technique, the compilation time tends to be analogous to the size of the system. One example of a call graph execution profiler for Ada is gprof [28].

- Aspects: Aspects can be deployed to monitor the execution of the program. In this approach, the locations of interest are defined through pointcut expressions. The efficiency and suitability of aspects for the purpose of tracing is demonstrated by [16, 29]. The advantages of this approach can be summarized as follows:

    1. Aspects are flexible since pointcuts can be highly expressive. They can monitor subprogram calls, executions, object initializations, deconstruction, etc.

    2. Due to the obliviousness[2] property in AOP, the code under analysis does not need to be modified or even be aware of being monitored.

    3. Aspects can obtain detailed information about reached join points, such as involved objects, invoked methods (subprograms), actual arguments and

[2]In [22] authors Filman and Friedman argued that obliviousness is a necessary characteristic for AOP. This property implies that functional components are unaware of the existence of aspectual behavior.

return values.

4. Profiling can be well localized into a single aspect definition.

2.6.2 Trace visualization techniques for dynamic analysis

In order to analyze execution traces, a representation model is required. There exist three main visualization techniques used to represent execution traces which are summarized below:

- Graph-based visualization: Typically a call graph is deployed to represent the interactions between subprograms at runtime. A call graph is a directed graph that represents call relations between subprograms in a computer program. In this graph, nodes representing the subprograms belonging to a specific package are connected to each other through edges representing a call from its source node to its target node.

- A modeling language: Typically, a Unified Modeling Language (UML) sequence diagram [59] is deployed to visualize the system's behavior.

- Text: Execution traces can be represented and stored as lists of events in plain text [28].

Chapter 3

Problem and motivation

In this chapter we discuss the problem and the motivation behind this research which constitutes the scope of this dissertation.

We have previously discussed that many old Ada systems are still operating and providing value to their stakeholders. However, due to different types of maintenance (particularly perfective and corrective) these systems tend to have gained weight and to have fallen into a legacy state. When falling into a legacy state, a system tends to exhibit the difficulties associated with software aging and its quality starts to degrade after performing perfective and corrective maintenance activities. One of the main quality attributes of the system which is directly affected by software aging is modularity. Based on the ISO 9126 model (Section 2.1), modularity is a pure internal attribute that influences many software characteristics and sub-characteristics. Therefore, in order to evaluate and improve the quality of a legacy system, we need to look into the level of modularity of its components. We can break the problem down into three main subproblems: 1) comprehension, 2) quality indicators and 3)

metrics, where each one is being described below:

Program comprehension The first challenge in dealing with legacy systems is obtaining information about either their structure or their behavior. Despite extensive research about comprehension of Ada programs [41, 57], few approaches have addressed system quality, and modularity in particular.

Modularity as a quality indicator The degraded modularity has motivated us to investigate this property in Ada programs. In order to be able to evaluate and improve program modularity, we need to have metrics for measurement purposes. As we mentioned in Section 2.3, coupling and cohesion of system packages can be indicators of the degree to which the system packages are modularized. Various metrics for measuring coupling and cohesion of Ada programs have been proposed in past investigations [9, 55, 56, 64]. However, all of them are based on a static analysis of the system. As discussed in Section 2.6, static analysis results in a huge amount of data [18]. While most of Ada programs are medium- to large-scale, static analysis will not be fully efficient, unless some abstraction models are applied. Furthermore, empirical studies have indicated that static analysis is not sufficient for capturing dynamic dependencies among system modules such as those related to polymorphism, dynamic binding and inheritance [24].

Metrics A lot of metrics are defined in the literature to capture software quality. However, they are mostly focusing on the software development process before product deployment. They are mainly used as a baseline or a threshold for decision-making

purposes about different factors such as cost estimation, defect prediction, risk minimization, performance optimization and user interface efficiency, but none of them are defined or used to evaluate the quality of an existing software for maintenance purposes.

These problems motivate us first to investigate comprehension of Ada programs and subsequently, to reason about the level of modularity of systems based on the obtained information.

Chapter 4

Proposal

In order to resolve the problems mentioned in Chapter 3, we propose two complementary techniques, illustrated in the UML activity diagram of Figure 3 and discussed subsequently.

Program comprehension We plan to deploy dynamic analysis by monitoring the execution traces of an Ada program. This way, we make sure that efficient and guaranteed information about dynamic behavior of the system has been generated. We store the obtained data from dynamic analysis of the system in a relational database for providing cumulative data. In order to provide a complementary view of system behavior, we plan to visualize it in the form of a call graph which can be considered as a dynamic model of the system interactions.

Reasoning about modularity We plan to perform fan-out analysis on the call graph, in order to come up with a set of new metrics and in some cases, adopt and

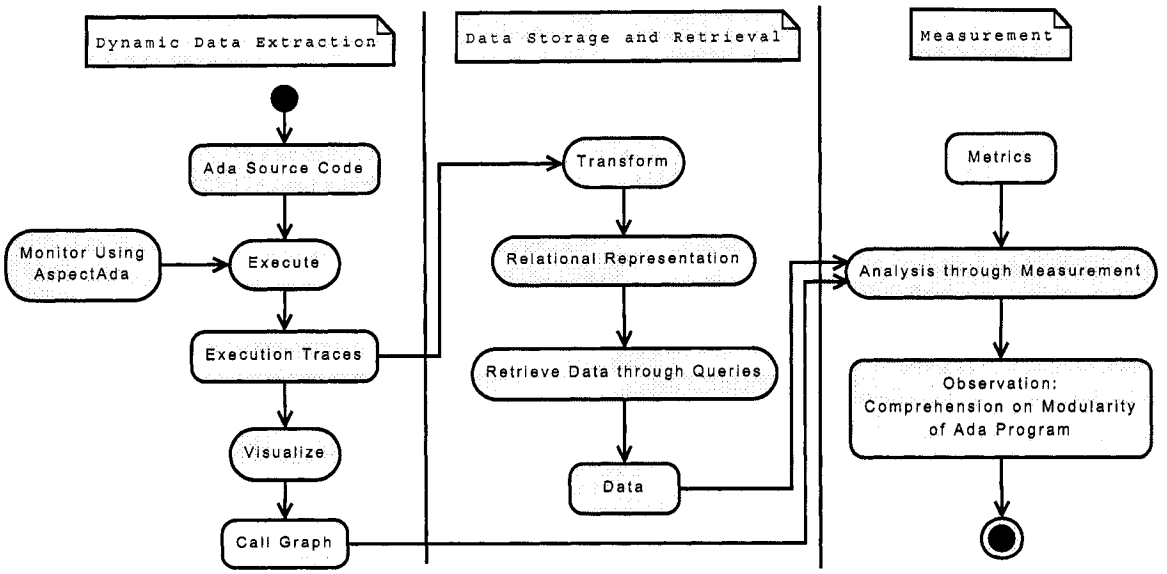

Figure 3: UML activity diagram illustrating dynamic analysis to provide comprehension.

refine existing metrics for quantitative measures of coupling, cohesion and modularity. Having such metrics allows us to define thresholds for each criteria and easily reason about the level of coupling, cohesion and modularity of any system under investigation.

The expected contributions of this proposal are to provide an environment for system maintainers to gain comprehension over the quality of legacy Ada programs through dynamic analysis and applying a set of validated metrics in order to measure modularity, coupling and cohesion of the system modules. Applying the proposed approach in this work, system maintainers can investigate how the proposed metrics behave as the system evolves. This way, they can get concrete indications on the degree to which maintenance tasks such as refactoring and reengineering improve the overall quality of the system.

Chapter 5

Program comprehension

As we discussed in Section 2.5, Zeller [65] came to the conclusion that most of the existing tools and techniques for system comprehension are using deduction and observation, while integrating experimentation with induction or deduction is one of the main challenges in program comprehension. However, the approach we are following in this work, is an integration of observation, induction and experimentation. Three main components forming our methodology are:

- Observation by profiling the execution of a program in order to obtain traces.
- Induction by summarizing the information extracted during observation and visualizing it in a call graph.
- Experimentation by running first the initial program and then the refactored program (to be discussed in Chapter 7) to investigate the improvement of quality attributes.

We provide comprehension of the program by deploying dynamic analysis, capturing the execution traces, storing the data and visualizing it to obtain better understanding of the dynamic events at runtime.

5.1 Deploying dynamic analysis

We apply dynamic analysis on an Ada program through the following steps:

1. Generate raw data: We execute the system through a specific use-case scenario and input values.

2. Extract data: We capture execution traces and store them into a relational database schema.

3. Analyze derived attribute values: We analyze the data through applying a set of metrics which we have defined or adopted from the literature.

As we mentioned in Section 2.6, the main drawback of dynamic analysis is that the result is based on specific scenarios and input values and cannot be generalized for other executions. Therefore, maintainers should make sure that the set of chosen scenarios fully covers all possible paths in the code. It may be impossible to find one scenario that fully covers all possible execution paths, but *cumulative results* can help to provide more precise analysis, since results obtained from execution of several scenarios can be accumulated and later be used for analysis purposes.

5.2 Representing dynamic information through execution traces

The result of dynamic analysis is typically represented in the form of execution traces. An execution trace is an ordered list of interactions between subprograms to run a specific scenario at runtime. The interactions can occur in different forms, such as access to a common attribute, modification of a common attribute, etc. However, in this work, an interaction between subprograms is narrowed down to only the call relations. In order to represent execution traces, we need to provide some definitions:

Definition 1: A message is a call from one subprogram (caller) to another subprogram (callee). Thus, it can be represented as:

$$Message ::< CallerPackage.CallerSubProgram, CalledPackage.CalledSubProgram >$$

Definition 2: An execution trace refers to a sequence of messages. The order corresponds to the timing order of occurrence of the messages.

$$ExecutionTrace :: (Message_0, Message_1, Message_2, \dots)$$

In this project, we concentrate on package level analysis, implying that during program execution we make no distinction between different objects of the same package. During program execution, we follow the convention of substituting all

active objects of the package with the name of the package.

5.3 Obtaining traces: Deploying AspectAda as an instrumentation tool

As we discussed in Section 2.6.1, there is a set of benefits associated with the use of aspects for tracing the system behavior. These benefits motivated us to deploy AspectAda to build a tracer for Ada programs. The tracer aspect, named `Tracer_Aspect`, is composed of a weaving rule file (.aaw), an aspect body file (.aab) and an aspect specification file (.aas). For the purpose of dynamic analysis based on interactions between system subprograms, `Tracer_Aspect` should be able to capture the execution of each subprogram. This join point model is realized through the pointcut definition in a composition rule file as:

```
weaver Tracing_Rules is

  Execution_PC : Pointcut := execution(*.*(..));
        ...

end Tracing_Rules;
```

In the aspect body file, `Tracer_Aspect` declares `Tracer_A` type as a `Detailed_Aspect` which provides more detailed information about the captured join point. This type defines three records as `Log_Count` in order to keep track of order of execution, `Exec_Depth` in order to keep track of nesting levels of the executions, and a file as `Output_File` as the destination output for storing the traces. Finally, one `around`

advice is defined in this file. The implementation of the specification file is as:

```
aspect  Tracer_Aspect is

  type Tracer_A is new Aspect_Ada.Detailed_Aspect with record

    Log_Count : Integer := 0;
    Exec_Depth: Integer := 0;

  end record;

  advice Around (Tracer : Tracer_A);

  Output_File : File_Type;
  Logger : Logger_A;

end Logger_Aspect;
```

The `Tracer_Aspect` body file provides the implementation for the `around` advice as following:

Before proceeding to the execution of the captured join point, the value of `Log_Count` and `Exec_Depth` increases by one. Also, the logging information is printed in the `Output_File` file. Following that, control flow returns to the captured join point and the subprogram runs. In order to form a hierarchy of nested subprograms execution, after proceeding and completing the execution of the subprogram, the value of `Exec_Depth` should be decreased by one.

Consequently, when an Ada program runs, upon reaching the execution of any subprogram, runtime information such as entering or exiting the called subprogram, containing package or type, subprogram arguments, and the sequence of execution will be saved in a text file. This text file will be parsed and reused later during program analysis and measurement (to be discussed in Section 5.4). A sample sequence of

```
aspect body Tracer_Aspect is

  advice Around (Tracer : Tracer_A) is begin

    Tracer.Log_Count:=Tracer.Log_Count+1;
    Tracer.Exec_Depth:=Tracer.Exec_Depth+1;

    Ada.Text_IO.Put_Line( Output_File, Aspect_Ada.Image
                          (Aspect_Ada.Get_Join_Point(Tracer).all)
                          & Integer'Image(Tracer.Log_Count)
                          & Integer'Image(Tracer.Exec_Depth));

    Proceed;
    Logger.Exec_Depth:=Logger.Exec_Depth-1;

  end Around;
end Tracer_Aspect;
```

traces provided by `Tracer_Aspect` is given in Table 1 and the hierarchy of calls of this set of traces is shown in Figure 4. It is necessary to mention that in Table 1, P_i is the notion of package and SP_j is the notion of the subprogram which is followed by the signature of the subprogram.

Table 1: Sample traces provided by `Tracer_Aspect`.

Call	Log_Count	Exec_Depth
P0.SP0 (To : in Position)	1	1
P0.SP1 (To : in Position)	2	2
P0.SP2 (To : in Position)	3	3
P1.SP3 (To : in Position)	4	3
P2.SP5 (To : in Position)	5	2
P2.SP6 (To : in Position)	6	3
P2.SP7 (To : in Position)	7	4
P1.SP4 (To : in Position)	8	4

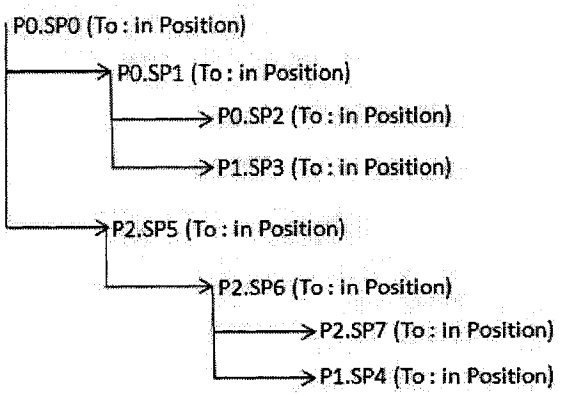

Figure 4: The hierarchy of subprogram calls.

$$ExecutionTrace :: (Message_0, Message_1, Message_2, Message_3, \dots) ::$$
$$(< P_0.SP_0, P_0.SP_1 >, < P_0.SP_1, P_0.SP_2 >,$$
$$< P_0.SP_1, P_1.SP_3 >, < P_0.SP_0, P_2.SP_5 >, \dots)$$

The structure and implementation of the tracer does not need to change when being applied on different Ada programs, since the tracer is not dependent on the context of the program. However as a limitation of this approach, we can mention the abstraction of execution traces, meaning that abstraction or selection of a subset of traces can be done only through AspectAda code. For choosing a specific hierarchy level of the execution or only message passing among a specific subset of packages in the system, the user (maintainer) can only do that by changing the code of instrumentation tool. It would be more flexible if AspectAda could provide the ability for the user (maintainer) to choose the abstraction or selection levels without manipulating the AspectAda code.

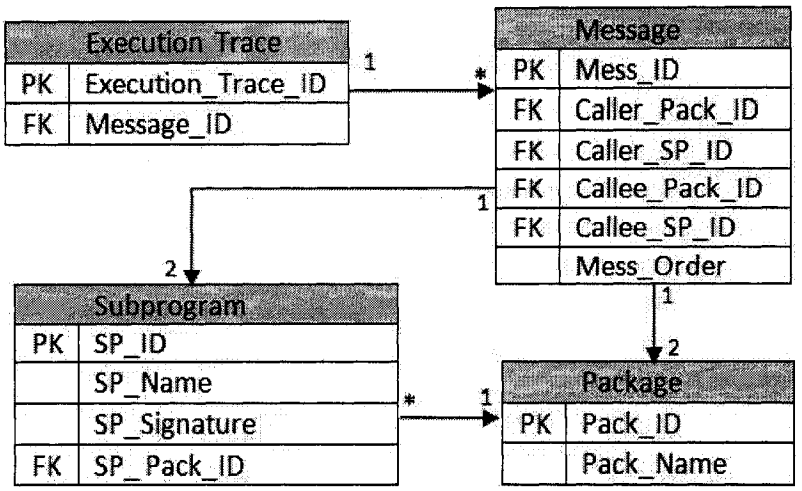

Figure 5: Relational database schema for storing execution traces.

5.4 Storing traces in a relational database

The obtained execution traces need to be kept in some data storage. One viable solution for such a data storage can be a relational database, a schema of which is shown in Figure 5. To implement this solution we can identify two options: The first option would be to have `Tracer_Aspect` dynamically creating and populating the database tables. However, this approach would considerably decrease the program performance, since the aspect would have to handle the connection and all other interactions with the database while the program is running. The second option would be to save the execution traces in a text file. A separate program would then read the text file, create the database, and populate its tables. We have used the second option in our approach for the purpose of performance issues.

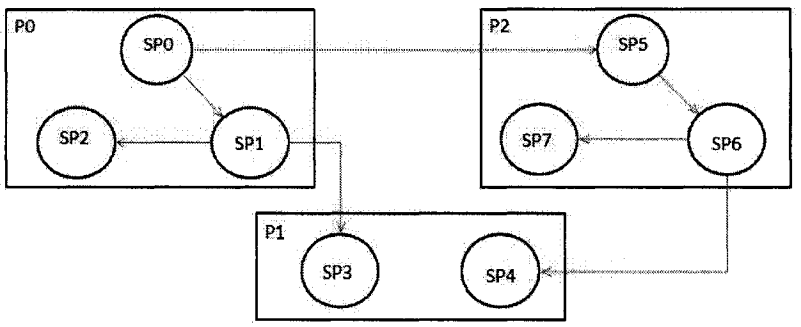

Figure 6: The call graph corresponding to the hierarchy of execution traces shown in Figure 4.

5.5 Visualizing dynamic information

In order to represent the dynamic behavior of the system, we deploy the notion of a *Call Graph*. A call graph G is defined as:

$$G = (E, R)$$

where E is a set of nodes that corresponds to single-entry-single-exit segments of code (in Ada programs, this is a "subprogram") and R is a set of messages, as a binary relation on members of E denoting a message passing from a source node to a target one.

Figure 6 illustrates the call graph corresponding to the sample traces in Section 5.3 and its corresponding hierarchy of subprograms call, shown in Figure 4. In this call graph,

$$E = \{\ P_0.SP_0, P_0.SP_1, P_0.SP_2, P_1.SP_3,$$
$$P_1.SP_4, P_2.SP_5, P_2.SP_6, P_2.SP_7\ \}$$

$$R = \{\ < P_0.SP_0, P_0.SP_1 >, < P_0.SP_1, P_0.SP_2 >,$$
$$< P_0.SP_1, P_1.SP_3 >, < P_0.SP_0, P_2.SP_5 >,$$
$$< P_2.SP_5, P_2.SP_6 >, < P_2.SP_6, P_2.SP_7 >,$$
$$< P_2.SP_6, P_1.SP_4 >\ \}$$

It is important to note that a call from a specific subprogram to another may occur more than once during the execution of one scenario; that is, the corresponding edge should appear more than once in the graph. Since this repetition would make the graph confusing and complex, the edge is drawn once. Since we are interested in evaluating the occurrence of call relations between subprograms, the number of occurrences will not add more value to our analysis, because the occurrences may take place during a loop statement, thus a high number of occurrences does not imply more coupling or less cohesion. However, the number of occurrences of this edge can be easily retrieved from the database. The same approach is followed for the order of occurrence of the message. Since the graph is not suitable for representing the order of message occurrence, this order is not shown in the call graph. However, this number is saved in the database for each edge and can be retrieved during analysis if and whenever required.

5.6 Scalability

The scale of Ada programs and the need for providing cumulative results raises the issue of scalability. Even though database management systems can solve the storage problem, there still exists another problem with visualization, due to the constraint of limited space. As an example, consider an Ada program with five packages where each package has an average of five subprograms. As a result, a full code coverage scenario or a cumulative analysis will produce a graph with almost $\binom{2}{5\times5} = 300$ edges, which makes a confusing and unreadable call graph. Therefore, in order to have a better visualization, some abstraction approaches should be considered. We can produce a more abstract view of a call graph by aggregating all nodes inside a package and represent them by a single node as the containing package. The edge between each two nodes is associated with a number that shows the summation of individual edges from any subprogram inside a source package to any subprogram inside the target package. This way, the number of edges between each pair of nodes will not be greater than two.

Some approaches for trace summarization and compression have been proposed [30, 31]. However, this issue is out of the scope of this research.

As the summary of this chapter, we have gained comprehension of the program by applying dynamic analysis through monitoring the program execution by AspectAda as an instrumentation tool, visualizing the structure and the behavior of the programs at runtime and finally storing the obtained information in a relational database for providing accumulative data and more precise analysis.

Chapter 6

Quality measurement

In order to have a template for definitions and representations, we adopt the notion of *modular system*, originally defined by Briand, Morasca, and Basili [10] as a 3-tuple $MS = < E, R, M >$, where E is a set of entities, R is a set of all relations between each pair of entities and M is a set of all modules in the system. We consider a call graph as a modular system, defined by a 3-tuple $MS = < E, R, M >$, where E is a set of all subprograms appear in the call graph (in the examined scenario), R is a set of all call relations between the subprogram, and M is a set of all packages to which the subprograms belong statically or are dynamically dispatched (in the case of polymorphism).

The call graph of a sample Ada program is shown in Figure 7. In this modular system, E, R and M are as follows:

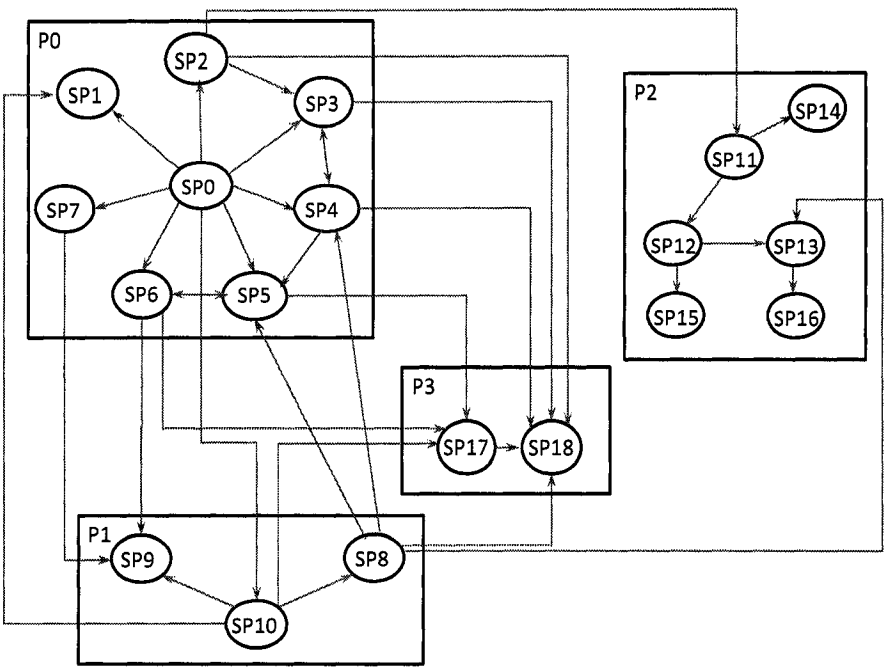

Figure 7: The call graph of a sample program shown as a modular system.

$$E = \{\ SP_0, SP_1, ..., SP_{18}\ \}$$

$$\begin{aligned} R = \{\ & < P_0.SP_0, P_0.SP_1 >, < P_0.SP_0, P_0.SP_2 >, \\ & < P_0.SP_0, P_0.SP_3 >, < P_0.SP_0, P_0.SP_4 >, \\ & < P_0.SP_0, P_0.SP_5 >, < P_0.SP_0, P_0.SP_6 >, \\ & \ldots, \\ & < P_2.SP_{12}, P_2.SP_{15} >, < P_2.SP_{13}, P_2.SP_{16} >\ \} \end{aligned}$$

$$M = \{\ P_0, P_1, P_2, P_3\ \}$$

Briand et al. [10] consider coupling as inter-module and cohesion as intra-module dependency in the modular system. Based on the call graph of an Ada program,

coupling between two packages is considered as the degree of dependency between these two packages which are connected through a call from one subprogram in one package to a subprogram in another package. In the same way, cohesion of a package is considered as the degree of dependency among all subprograms within the same package. In order to separate the measurement of coupling and cohesion based on the call graph, we adopt the notions of *Intermodule-Edges Graph* and *Intramodule-Edges Graph* mentioned by Allen et al. in [2]. According to these notions, the call relations are partitioned based on whether or not their caller and callee subprograms are placed in the same package. This implies:

$$R = R_{inter} \cup R_{intra}$$

where,

R_{intra}: Set of edges where caller and callee subprograms are in one package.

R_{inter}: Set of edges where caller and callee subprograms are in two different packages.

An edge in the call graph either belongs to R_{inter} or R_{intra}, implying that:

$$R_{inter} \cap R_{intra} = \varphi$$

We can now split the call graph into two subgraphs: *Intra-Package Graph* and *Inter-Package Graph.* Each of these subgraphs has the same E as the set of subprograms, the same M as the set of packages, but a subset of R as the set of call relations between subprograms.

Definition 3: Having a call graph as a modular system MS, Intra-Package Graph, named MS_{intra}, is a subgraph of MS where,

$$MS_{intra} = < E, R_{intra}, M >$$

Figure 8 is the Intra-Package graph of the call graph shown in Figure 7.

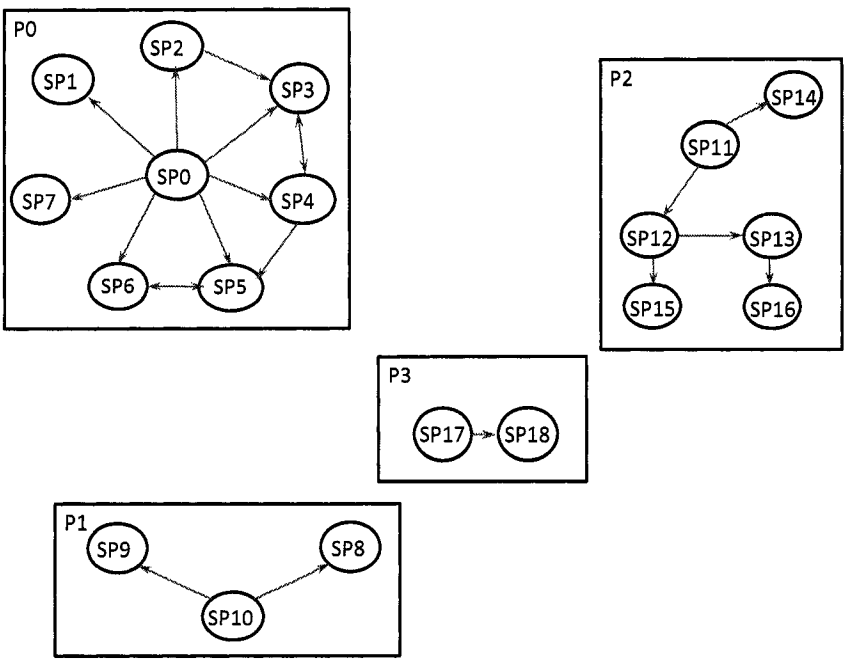

Figure 8: Intra-Package Graph.

Definition 4: Having a call graph as a modular system MS, Inter-Package Graph, named MS_{inter}, is a subgraph of MS where,

$$MS_{inter} = < E, R_{inter}, M >$$

Figure 9 is the Inter-Package graph of the call graph shown in Figure 7.

In the next sections, we define metrics to measure coupling and cohesion. Before introducing these metrics, we need to consider the following definitions:

Definition 5: Given a package P_i, $fan\_out(P_i, P_j)$ is defined as the number of calls sent by package P_i to package P_j in the call graph.

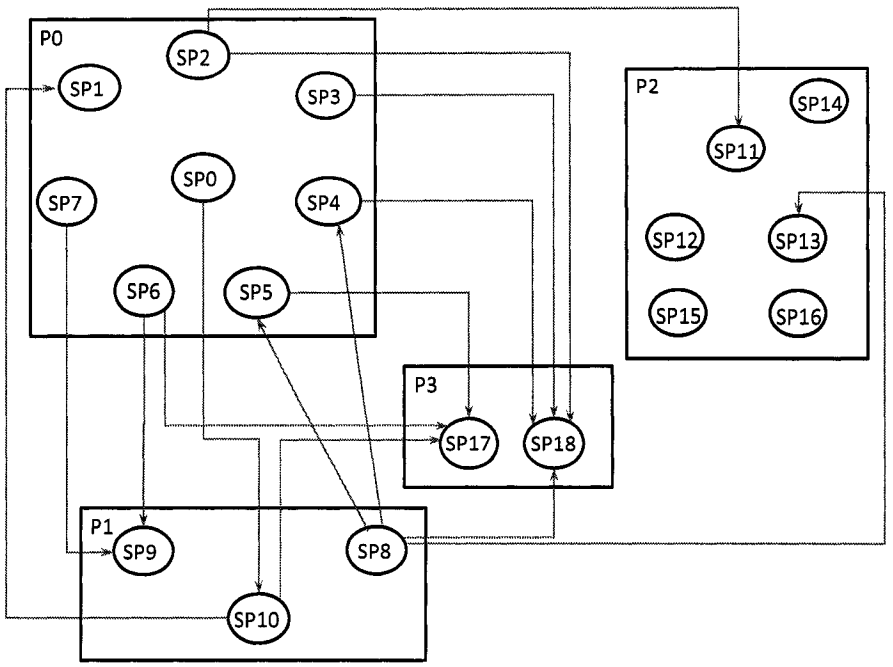

Figure 9: Inter-Package Graph.

By the following query, the value for $fan\_out(P_i, P_j)$ can be directly retrieved from the database[1]:

```
select count(msg.Mess_ID)
from Message msg join Package pckg1 join Package pckg2
  on msg.Caller_Pack_ID = pckg1.Pack_ID and
  msg.Callee_Pack_ID = pckg2.Pack_ID
where
  pckg1.Pack_Name = "Pi" and
  pckg2.Pack_Name = "Pj" and
  pckg2.Pack_Name <> "main";
```

Definition 6: Given a subprogram SP_i, $fan\_out(SP_i, SP_j)$ is defined as the

[1]In the database, "main" as the `Pack_Name` and "main" as the `SP_Name` refers to the call made by the system to the main file.

number of call relations sent by subprogram SP_i to subprogram SP_j in the call graph.

The following query calculates the value for $fan\_out(SP_0, SP_1)$ directly from the database:

```
select count(msg.Mess_ID)
from Message msg join Subprogram sbpg1 join Subprogram sbpg2
  on msg.Caller_SP_ID = sbpg1.SP_ID and
  msg.Callee_SP_ID = sbpg2.SP_ID
where
  sbpg1.SP_Name = "SP0" and
  sbpg2.SP_Name = "SP1" and
  sbpg2.SP_Name <> "main";
```

It is important to mention that the number of occurrences of a call is not important in measuring coupling and cohesion, since a subprogram can be invoked inside a loop statement, thus it does not imply stronger dependency. In the subsequent discussion on metrics, we are only concerned about the number of individual edges from/to a package.

6.1 Cohesion

In an Ada program, cohesion is the degree to which subprograms in a package are related to implement a single functionality.

It is necessary to note that in the literature there are two types of cohesion: *Structural cohesion* and *conceptual cohesion.* Structural cohesion deals with the physical

connection between the elements of a design component and investigates how tightly the attributes and operations are encapsulated in a module. Conceptual cohesion focuses on how the elements of a module are conceptually related and collaborate to implement a single functionality/concern.

We believe that the more the subprograms inside a package invoke each other rather than invoking subprograms from other packages, the more they are logically related to implement a single concern without depending on other packages. This implies that strong relations through subprogram invocation inside a package (structural cohesion) lead us to conclude that the package is conceptually cohesive.

Xu et al. in [64] have defined a metric to measure the cohesion of a package in Ada programs. In their approach, three factors affecting the cohesion are defined as object-object, subprogram-object and subprogram-subprogram cohesion. The integration of these three factors is considered as cohesion of the package. In this research, we adopt and refine the third factor as subprogram-subprogram cohesion to be consistent with our call graph.

Definition 7: Given a package P_x with n subprograms $SP_0, SP_1, ..., SP_{n-1}$, the cohesion of P_x, $P\_Cohesion(P_x)$, is defined as:

$$P\_Cohesion(P_x) = \begin{cases} 1 & n = 1 \\ \frac{1}{n} \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \frac{Relation(SP_i, SP_j)}{n-1} & n > 1 \end{cases}$$

$$Relation(SP_i, SP_j) = \begin{cases} 0 & fan\_out(SP_i, SP_j) = fan\_out(SP_j, SP_i) = 0 \\ 1 & others \end{cases}$$

In these formulas, we consider $fan\_out(SP_i, SP_j) = 0$ when $i = j$, and as the result $Relation(SP_i, SP_j) = 0$. The reason is that a call from one subprogram to itself does not increase the cohesion of the containing package. $Relation(SP_i, SP_j)$ defines whether or not two subprograms are related to each other through calling or being called by each other. This way, the direction of the call can be ignored, since having a bidirectional call does not imply more relatedness. Overall, this metric calculates the ratio of the number of existing edges inside a package to the number of potential edges in the same package.

The value of $P\_Cohesion(P_x)$ can be calculated directly from the database by the following query:

```
select 2*(X.allrel - Y.repit/2) / (z.number*(z.number-1))
from
 (select count(*) as allrel
  from Message msg join Package pckg1 join Package pckg2
    on msg.Callee_Pack_ID = pckg1.Pack_ID and
```

```
  msg.Caller_Pack_ID = pckg2.Pack_ID
 where
  pckg1.Pack_Name = "Px" and
  pckg2.Pack_Name = "Px" and
  pckg2.Pack_Name <> "main" ) X,
(select count(*) as repit
 from
  (select msg.Caller_SP_ID as c1, msg.Callee_SP_ID as c2
   from Message msg join Package pckg1 join Package pckg2
    on msg.Callee_Pack_ID = pckg1.Pack_ID and
    msg.Caller_Pack_ID = pckg2.Pack_ID
   where
    pckg1.Pack_Name = "Px" and
    pckg2.Pack_Name = "Px" and
    pckg2.Pack_Name <> "main") msg1
 join
  (select msg.Caller_SP_ID as c3, msg.Callee_SP_ID as c4
   from Message msg join Package pckg1 join Package pckg2
    on msg.Callee_Pack_ID = pckg1.Pack_ID and
    msg.Caller_Pack_ID = pckg2.Pack_ID
   where
    pckg1.Pack_Name = "Px" and
    pckg2.Pack_Name = "Px" and
    pckg2.Pack_Name <> "main") msg2
```

```
  where c1 = c4 and
  c2 = c3) Y,
 (select count(*) as number
  from subprogram sbpg join Package pckg
    on sbpg.SP_Pack_ID = pckg.Pack_ID
  where
pckg.Pack_Name = "Px") Z;
```

Definition 8: Given a system with m packages $P_0, P_1, ..., P_{m-1}$, the cohesion of the system, $S\_Cohesion$, is defined as:

$$S\_Cohesion = \frac{\sum_{i=0}^{m-1} P\_Cohesion(P_i)}{m}$$

The value of $S\_Cohesion$ can be derived by having the cohesion of each package and the value of m. Therefore, combining the above mentioned query and the following query for obtaining the value of m, $S\_Cohesion$ can be calculated:

```
select count(*)-1 from Package;
```

We consider the average cohesion of all packages as the cohesion for the overall system. The obtained value of $S\_Cohesion$ can be used to compare the overall cohesion of the system before and after applying a change to the system.

6.1.1 Analysis

Based on the proposed formula, if there is only one subprogram inside the package, the cohesion is equal to its maximum value: 1. This would imply that the subprogram

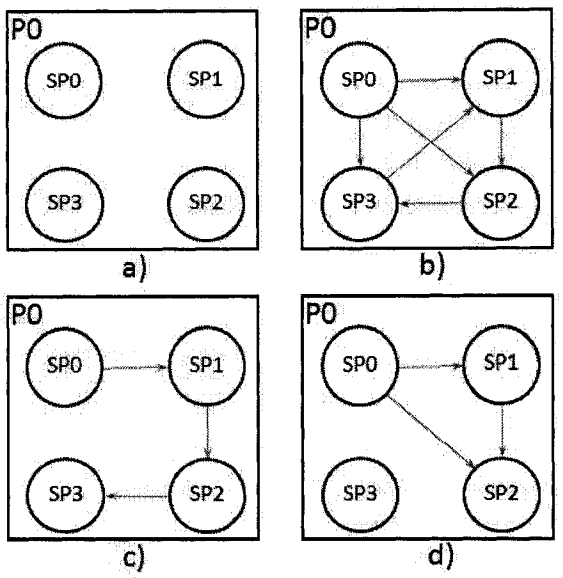

Figure 10: Package with a) minimum, b) maximum, c) $n-1$ and d) $\binom{2}{n-1}$ number of edges.

implements all the functionality of the package, so it holds the maximum degree of relatedness.

Based on this formula, the value of cohesion for a package belongs to the interval $[0, 1]$. The formula implies that the more edges inside a package exist, the more cohesive the package is. While mapping the call graph of the package under analysis to a mathematical graph, each node represents a subprogram and each edge represents a call invocation from a source subprogram to a target subprogram.

In order to define a threshold for the cohesion of a package, two main cases should be considered, the worst and the best cases for the cohesion value.

The worst case occurs when all nodes inside the graph are isolated nodes. Figure 10.a represents a sample package with the minimum value of cohesion, that is $P\_Cohesion(P_0) = 0$.

The best case occurs when there is at least one edge between each pair of nodes

in the graph or the graph is fully connected. Figure 10.b represents a sample package with the maximum value of cohesion, that is $P\_Cohesion(P_0) = 1$.

For having a reasonable value of cohesion, the graph should be at least connected, meaning that all subprograms inside the package are being called or call other subprogram inside the same package. Having a node not connected to the graph indicates that this subprogram is not related to the functionality of the other subprograms inside the package.

We can consider the following cases where the graph is not connected:

Case 1: The graph is connected through the minimum number of edges, implying that by omitting an edge, the graph becomes disconnected (see Figure 10.c).

Representing the call graph of package P_i by a graph $G = < E, R >$ with $|E| = n$, the minimum number of edges that keeps the graph connected is $n - 1$. Thus,

$$if\ |R| < n - 1$$

$$\Rightarrow \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} Relation(SP_i, SP_j) < 2(n-1)$$

$$\Rightarrow P\_Cohesion(P_i) < \frac{2}{n}$$

Therefore, holding the cohesion value less than $\frac{2}{n}$ for a package with n subprograms implies that at least one of the subprograms inside the package is not logically related to the others. This value can be considered as the threshold for cohesion of a package. Holding the cohesion value less than $\frac{2}{n}$ implies that the graph is not connected. However, holding the cohesion value equal or greater than $\frac{2}{n}$ does not

necessarily imply that the graph is connected (explained in case 2).

Case 2: The number of edges is greater than or equal to $n-1$. However, some of them connect a pair of nodes which are connected through some other nodes and still the graph is disconnected. Figure 10.d represents this case where the number of edges is equal to $n-1$, but the graph is disconnected.

Representing the call graph of package P_i by a graph $G =< E, R >$ with $|E| = n$, the maximum number of edges that makes the graph connected is $\binom{2}{n-1}$. Thus,

$$if\ |R| > \binom{2}{n-1}$$

$$\Rightarrow \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} Relation(SP_i, SP_j) > (n-2)(n-1)$$

$$\Rightarrow P\_Cohesion(P_i) > 1 - \frac{2}{n}$$

Therefore, holding the cohesion value more than $1 - \frac{2}{n}$ for a package with n subprogram implies that all subprograms inside the package are logically related to each other. This value can be considered as the lower bound of having a high cohesion for the package. Thus, we can conclude that $\forall n > 3$,

$$Cohesion\ Level = \begin{cases} Low & P\_Cohesion(P_i) \in [0, \frac{2}{n}) \\ Medium & P\_Cohesion(P_i) \in [\frac{2}{n}, 1 - \frac{2}{n}] \\ High & P\_Cohesion(P_i) \in (1 - \frac{2}{n}, 1] \end{cases}$$

Table 2: Cohesion measure of the packages shown in the call graph of Figure 9.

Package	Cohesion	n	Cohesion Level
P_0	0.4	8	Medium
P_1	0.66	3	High
P_2	0.33	6	Medium
P_3	1	2	High
System	0.59	–	–

Deploying the formula as a metric, we can measure the cohesion of packages shown in Figure 9 and the results of these measurement are listed in Table 2. The results indicate that $P\_Cohesion(P_3) > P\_Cohesion(P_1) > P\_Cohesion(P_0) > P\_Cohesion(P_2)$. Even though these measures are relative to all packages in the system, we can conclude that subprograms in P_3 are more related than those in other packages, also the implemented concerns by package P_3 is more well-encapsulated than the concerns implemented by other packages. Based on this formula, package P_2 is the least cohesive package in this system. In the call graph shown in Figure 9, the ratio of existing edges to potential (possibly existing) edges is very low, implying that the subprograms do not strongly collaborate inside the package.

6.1.2 Formal validation

Briand et al. in [10] have discussed a set of properties to validate any cohesion measure. We can prove that our proposal for a cohesion metric can be validated, by applying all these properties as follows:

1. Non-negativity and normalization. As mentioned in the previous subsection, cohesion of a package belongs to the interval $[0, 1]$.

2. Null value. Cohesion of a package is equal to 0, if there is no Intra-Package dependency among its subprograms, because in this case the fan_out of all subprograms becomes 0.

3. Monotonicity. Adding an Inter-Package edge to a package does not decrease the cohesion of the package, since for calculating cohesion we are only concerned with Intra-Package edges.

4. Merging packages. If two packages, P_i and P_j for which there is no Inter-Package dependency between them, are merged to form a new package P_k, then the package cohesion of P_k would not be greater than the maximum of the package cohesion of P_i and P_j. The reason is that not only increasing the number of some unrelated subprograms in a package will not increase the cohesion, but it will also degrade the level of relatedness of subprograms inside the package.

6.2 Coupling

In an Ada program, coupling is the degree to which the functionality of a package is dependent on the operation of other packages. Thus, we use the Inter-Package graph to measure the coupling of a package to other packages, as in this graph only edges which are connecting two different packages are shown.

We define a metric to measure the coupling of a package to any other packages in the system and the system as a whole, based on the degree of dynamic dependency (call relation) from the given package to other packages.

Definition 9: Given a system with m packages $P_0, P_1, ..., P_{m-1}$, the coupling of package P_i to P_j, $P\_Coupling(P_i, P_j)$ is defined as:

$$P\_Coupling(P_i, P_j) = fan\_out(P_i, P_j)$$

In this formula, we consider $fan\_out(P_i, P_j) = 0$ when $i = j$, and as a result $P\_Coupling(P_i, P_j) = 0$, because if there exist any call from one package to itself, the related edge would appear in the Intra-Package graph, not the Inter-Package graph. The reason of choosing fan-out as the coupling indicator is that it shows how much a package sends messages to other packages and, as a result, how much its functionality is coupled to the functionality of other packages. Fan-in analysis for a package indicates how much other packages in the system are coupled to the mentioned package. Based on this metric, the same query used for calculating $fan\_out(P_i, P_j)$, mentioned in Definition 5, can be used for calculating the value of $P\_Coupling(P_i, P_j)$ directly from the database.

We have defined three other metrics which can provide some useful information about the system.

Definition 10: The coupling of a package P_i to all other packages or the system as a whole, $P\_Coupling(P_i, system)$, is defined as:

$$P\_Coupling(P_i, system) = \sum_{j=0}^{m-1} P\_Coupling(P_i, P_j)$$

The value of $P\_Coupling(P_i, system)$ can be calculated directly from the database by the following query:

```
select count(msg.Mess_ID)
from Message msg join Package pckg1 join Package pckg2
  on msg.Caller_Pack_ID = pckg1.Pack_ID and
  msg.Callee_Pack_ID = pckg2.Pack_ID
where
  pckg1.Pack_Name = "Pi" and
  pckg2.Pack_Name <> "Pi" and
  pckg2.Pack_Name <> "main";
```

Definition 11: The coupling of the system as a whole to a specific package P_j, $P\_Coupling(system, P_j)$, is defined as:

$$P\_Coupling(system, P_j) = \sum_{i=0}^{m-1} P\_Coupling(P_i, P_j)$$

The value of $P\_Coupling(system, P_j)$ can be calculated by the following query:

```
select count(msg.Mess_ID)
from Message msg join Package pckg1 join Package pckg2
  on msg.Callee_Pack_ID = pckg1.Pack_ID and
  msg.Caller_Pack_ID = pckg2.Pack_ID
where
  pckg1.Pack_Name = "Pj" and
  pckg2.Pack_Name <> "Pj" and
  pckg2.Pack_Name <> "main";
```

Definition 12: The coupling of the system, $S\_Coupling$, is defined as:

$$S\_Coupling = \frac{\sum_{i=0}^{m-1} \sum_{j=0}^{m-1} P\_Coupling(P_i, P_j)}{m}$$

The query to calculate *S_Coupling* is defined as follows:

```
select count(distinct msg.Mess_ID)/
      (count(distinct pckg.Pack_ID)-1)
from Message msg join Package pckg
where
  msg.Caller_Pack_ID <> msg.Callee_Pack_ID and
  msg.Caller_Pack_ID <> 1;
```

We consider the average coupling of all packages as the coupling for the overall system. The obtained value of *S_Coupling* can be used to compare the overall coupling of the system before and after applying a change to the system.

6.2.1 Analysis

Based on this Definition 10, if there is only one package in the system, coupling of the package would get its minimum value which is 0 since the package is not dependent on any other packages. However, it does not necessarily imply a good system design since this situation would indicate that the responsibilities in the system are not well-distributed, but this problem can reveal itself by measuring the package cohesion.

As we did for cohesion, in order to define a threshold for the coupling of a package, two main cases should be considered, the worst and the best cases for coupling.

The worst case occurs when each of the subprogram in the package send a message

to every other subprogram in the system, but this case occurs very rarely and it implies a very weak design of the system. Therefore, we consider the worst case as the situation where each of the subprogram in the package is communicating with all other packages in the system, implying that each subprogram sends at least one message to every other package. Figure 11.a represents a sample system call graph with the worst case of coupling[2]. Let n_i be the number of subprograms inside package P_i in a system with m packages, thus $\forall j \in \mathbb{N}, j = 0..m-1, m > 1$,

$$P\_Coupling(P_i, P_j) = n_i$$

$$P\_Coupling(P_i, system) = n_i \times (m-1)$$

As we mentioned, the case where $P\_Coupling(P_i, system) > n_i \times (m-1)$ is very unlikely to occur. We name this situation having a coupling *greater than the threshold.* However, we do not take this case into consideration.

The best case occurs when only one subprogram in the package is communicating with at most one of the other packages (subprogram inside the other package) in the system, implying that each package sends at most one message to one other package. Figure 11.b represents a sample system call graph with the best case of coupling.

Thus, $\exists! j \in \mathbb{N}, j = 0..m-1, m > 1$, and

[2]To improve clarity, subprograms inside packages are not shown in Figure 11.

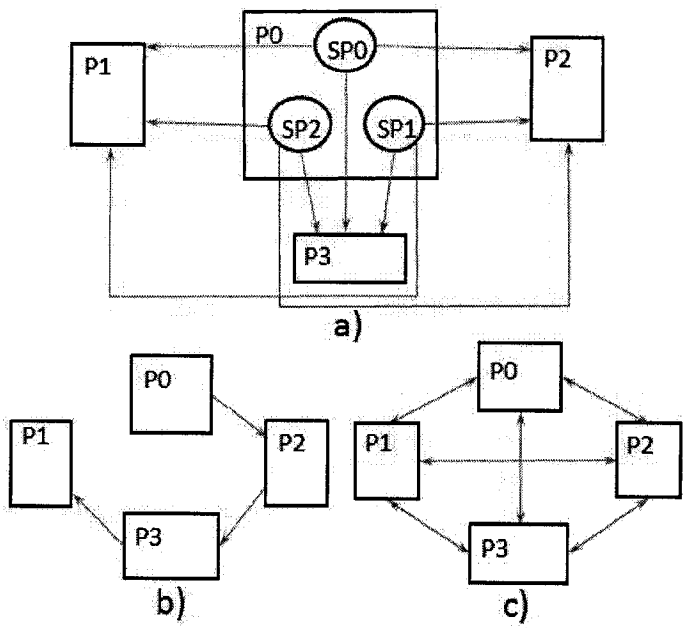

Figure 11: Package with the a) maximum, b) minimum and c) medium level of coupling.

$$P\_Coupling(P_i, P_j) = 1$$

$$P\_Coupling(P_i, system) = 1$$

The intermediate case occurs when only one subprogram inside a package communicates with more than one package in the system, that is considering each pair of packages of the system, there is only one edge from the source package to the target package. Figure 11.c is a sample system call graph representing this case. Thus, $\forall j \in \mathbb{N}, j = 0..m-1, m > 1$

$$P\_Coupling(P_i, P_j) = 1$$

$$P\_Coupling(P_i, system) = m - 1$$

Table 3: Coupling measure of the packages shown in the call graph of Figure 8.

Coupling of	P_0	P_1	P_2	P_3	System	Coupling Level
P_0	0	3	1	5	9	Strong
P_1	3	0	1	2	6	Strong
P_2	0	0	0	0	0	Weak
P_3	0	0	0	0	0	Weak
System	3	3	2	7	3.75	–

We consider this case as the threshold for the coupling value for a package when $m > 1$. That is, the possible levels for coupling will be based on the following,

$$Coupling\ Level = \begin{cases} Weak & P\_Coupling(P_i, system) \in [0, 1] \\ Medium & P\_Coupling(P_i, system) \in (1, m-1] \\ Strong & P\_Coupling(P_i, system) \in (m-1, n(m-1)] \end{cases}$$

According to this metric, the coupling measures of packages in the sample call graph shown in Figure 8 are listed in Table 3. As shown in Table 3, package P_0 is strongly coupled to P_3, P_1 and P_2. As a result, its level of coupling to the system is higher than the level of coupling between any other package and the system, i.e.

$$P\_Coupling(P0, System) > P\_Coupling(P1, System)\ and$$

$$P\_Coupling(P0, System) > P\_Coupling(P2, System)\ and$$

$$P\_Coupling(P0, System) > P\_Coupling(P3, System)$$

Conversely P_3 and P_2 are not coupled to other packages, while most of the packages are coupled to them. Having zero value for fan_out, they have the minimum amount

of coupling to the system which is 0.

6.2.2 Formal validation

Briand et al. in [10] have provided a set of properties to validate any coupling measure. We can prove that our proposal for a coupling metric can be validated, by applying all these properties as follows:

1. Non-negativity. In the above mentioned metrics (Definitions 9 and 10), the coupling of a package always has a non-negative value, because:
   $\forall P_i \wedge P_j,\ fan\_out(P_i, P_j) \geq 0.$

2. Null value. Based on the defined metric, when there is no call from a package to any other package in the system, it implies that $P\_Coupling(P_i, system) = 0$.

3. Monotonicity. Adding an Intra-Package edge to a package does not decrease coupling of the package, since in calculating coupling, we are only concerned about Inter-Package edges.

4. Merging packages. If two packages, P_i and P_j, are merged to form a new Package P_k, then the package coupling of P_k would not be greater than the summation of coupling values of P_i and P_j. The reason is that, the edges between P_i and P_j (if exist) which were considered as Inter-Package edges, become Intra-Package edges after merging. Therefore, the coupling of the newly formed package P_k will be less than or equal to the summation of P_i and P_j coupling values.

5. Disjoint package additivity. If two packages, P_i and P_j, for which there is no

Inter-Package dependency, are merged to form a new package P_k, then the package coupling of P_k would be equal to the summation of coupling values of P_i and P_j. The reason is that none of the edges from the previous Inter-Package graph will be considered as an Intra-Package edge. Thus, the coupling of the newly formed package P_k will be exactly equal to the summation of coupling values of P_i and P_j.

6.3 Modularity

As we mentioned in Section 2.3, cohesion and coupling are the dominant driving factors of modularity [17, 46, 54]. Therefore, in order to define a metric for system or package modularity both cohesion and coupling should be considered. Strong coupling for a package reduces its modularity, while cohesive packages tend to be more modularized. Based on these relations between modularity, cohesion and coupling, we define the modularity of a package as follows:

Definition 13: The modularity of a package P_i, $P\_Modularity(P_i)$, is defined as:

$$P\_Modularity(P_i) = \frac{P\_Cohesion(P_i)}{P\_Coupling(P_i, system) + 1}$$

Definition 14: The modularity of the system, $S\_Modularity$, is defined as:

$$S\_Modularity = \frac{S\_Cohesion}{S\_Coupling + 1}$$

Table 4: Critical cases for measuring the modularity of package P_i.

Case	$P\_Cohesion(P_i)$	$P\_Coupling(P_i, system)$	$P\_Modularity(P_i)$
1	0	0	0
2	0	$n(m-1)$	0
3	1	0	1
4	1	$n(m-1)$	$\frac{1}{n(m-1)+1}$

The reason behind incrementing the coupling measure by 1 in the denominator of these two metrics is to avoid division by zero. The value of $P\_Modularity(P_i)$ and $S\_Modularity$ can be retrieved from the database by merging the queries of $P\_Cohesion(P_i)$ and $P\_Coupling(P_i, system)$. The query is not shown here due to space limitations.

6.3.1 Analysis

Depending on the boundary values for $P\_Cohesion(P_i)$ and $P\_Coupling(P_i, system)$ one of the cases (rows) of Table 4 may occur.

Case 1: It occurs when the cohesion of the package has its minimum value, implying that none of the subprograms inside the package are connected together, but instead they are only communicating with other packages. At the same time, package coupling is at its minimum. This case implies that the subprograms inside this package are called by other packages and they are more strongly related to the functionalities implemented in other packages rather than those in P_i. This module should be broken down and each subprogram should be localized in the most related package. In this case, we define the minimum value for modularity as $P\_Modularity(P_i) = 0$.

Case 2: This case is the worst case where the subprograms inside the package are

not related together and at the same time they are highly coupled to functionalities of other packages. This module should be broken down and each subprogram should be placed in a package to which it is the most related. Based on the proposed metric for measuring modularity, $P\_Modularity(P_i) = 0$.

Case 3: This case is the best case where the package holds the maximum value for cohesion and the minimum value for coupling, implying that P_i is a cohesive package with no dependency on other packages. Based on the proposed metric for measuring modularity, $P\_Modularity(P_i) = 1$.

Case 4: It occurs when the subprograms inside the package are strongly related, but they are highly coupled to other packages in the system. The best solution for this case is to keep this cohesive package P_i, and transfer the subprograms of other packages to which P_i is highly coupled. Based on Definition 13, $P\_Modularity(P_i) = \frac{1}{n(m-1)+1}$.

If the package holds any other value for its coupling or cohesion, its modularity can be calculated by $P\_Modularity(P_i)$ (Definition 13).

Based on these four cases, we can conclude that for a given package P_i,

$$0 \leq P\_Modularity(P_i) \leq 1$$

However, we can define a threshold for modularity of a package based on all possible values for cohesion and coupling of a package (shown in Figure 12). We define the best, medium and worst ranges of modularity of a package based on obtained ranges of coupling and cohesion of the package. As shown in Figure 12 for both cohesion

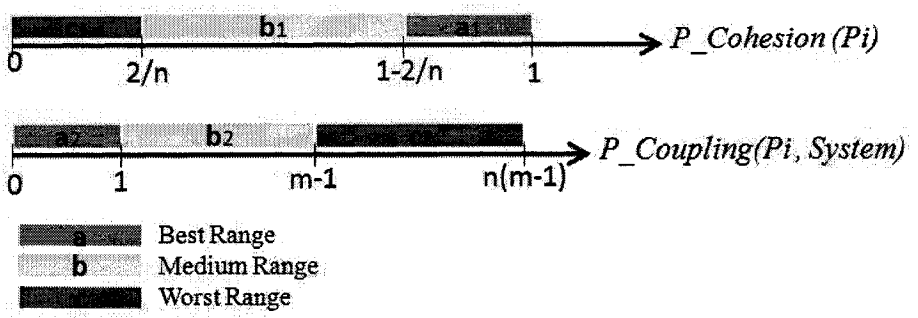

Figure 12: Possible values for cohesion and coupling of a package.

and coupling, the a, b and c ranges represent the best, medium and worst ranges respectively. We consider the best case of modularity when both cohesion and coupling are in best range, the worst case of modularity when both cohesion and coupling are in worst range, and the medium case of modularity for any other situations. That is,

$$Modularity\ Level = \begin{cases} High & P\_Cohesion(P_i) \in [1 - \frac{2}{n}, 1]\ and \\ & P\_Coupling(P_i, system) \in [0, 1] \\ Low & P\_Cohesion(P_i) \in [0, \frac{2}{n})\ and \\ & P\_Coupling(P_i, system) \in (m-1, n(m-1)] \\ Medium & Others \end{cases}$$

Since modularity is a ratio of two factors, we cannot define a threshold for modularity based on a specific value, and as a result we cannot anticipate a range of values for intermediate situations where the range of coupling and cohesion are not the same. Based on Figure 12, these intermediate situations occur when the package holds one of these pairs for its cohesion and coupling values respectively; (a1,b2),

Table 5: Modularity measure of the packages in the call graph shown in Figure 7.

Modularity of	Cohesion Level	Coupling Level	Modularity Level	Modularity Value
P_0	Medium	Strong	Medium	0.04
P_1	High	Strong	Medium	0.09
P_2	Medium	Weak	Medium	0.33
P_3	High	Weak	High	1
System	–	–	–	0.124

(a1,c2), (b1,a2), (b1,b2), (b1,c2), (c1,a2) or (c1,b2). Thus, the level of modularity of a package should be determined based on the level of corresponding measures for cohesion and coupling. However, the calculated number for modularity based on Definition 13 can be used for comparing the modularity of different packages, or the modularity of the package before and after restructuring or any other change during the maintenance phase.

Table 5 shows the modularity of the packages of the call graph shown in Figure 7.

Based on the calculated values for modularity, we can conclude that

$P\_Modularity(P_0) < P\_Modularity(P_1) < P\_Modularity(P_2) < P\_Modularity(P_3)$.

As a summary of this chapter, we have defined measures for cohesion, coupling and modularity and related thresholds for reasoning about the modularity of packages in the system and as the result find the packages whose modularity as one of the main internal attributes has been corrupted.

Chapter 7

Validation through a case study

In this chapter, we take a case study, then we monitor the program execution using AspectAda. This experience proved how efficiently the dynamic behavior of the system can be obtained without modifying the source code, especially in cases where the system is medium- to large-scale. Then, we apply the approaches mentioned in Chapter 5 to obtain comprehension over the dependencies between system modules, and finally we use the metrics defined in Chapter 6 to measure the modularity of the system. In order to validate the proposed metrics, we apply certain refactorings on the system, expecting an improvement in the cohesion, coupling and modularity of the system. Then, by measuring each of these factors in the refactored system, we check whether or not the obtained result is the same as what we are expecting. In this way, we ensure whether or not our investigation on "how the proposed metrics behave as the system evolves" as one of the expected contributions of this research is accomplished.

We apply our approach on an open source project called *Shapes* which is in-

Table 6: Captured traces of Shapes project.

Call	Log_Count	Exec_Depth
Screen.Put_Point (: Integer)	78	3
Shapes.Lines.Draw (: in Line_Type)	79	2
Screen.Put_Line (: Point_Type)	80	3
Screen.Put_Line (: Integer; : Integer)	81	4
Screen.Put_Point (: Integer)	82	5
Screen.Put_Point (: Integer)	83	5
Shapes.Lines.Draw (: in Line_Type)	84	2
Screen.Put_Line (: Point_Type)	85	3
Screen.Put_Line (: Integer; : Integer)	86	4

cluded in the GNAT package [27]. In this system, there are three defined types as Line_Type, Rectangle_Type and Face_Type where all three are the extended types of Shape_Type. They inherit some of the subprograms and introduce new subprograms. The scenario under which the system is being analyzed is as follows: first the user creates some shapes such as line, rectangle or face (which is implemented by initialization subprograms), then he refreshes the screen by clearing and drawing these shapes. The user can also move each shape; after each move, the screen gets refreshed automatically. Also, the user can place the existing shapes to create different figures on the screen (which is implemented by stack subprogram) and this action is followed by refreshing and drawing the figures automatically. We run the program and execution traces are captured by Logger_Aspect. Table 6 lists a partial sequence of the extracted traces.

The numbers in Log_Count column are the order of the calls and the numbers in Exec_Depth column correspond to the level of the calls in the overall system call hierarchy. After the traces are extracted, we populate the corresponding tables in the database to have cumulative data for further analysis. Having the traces, we can

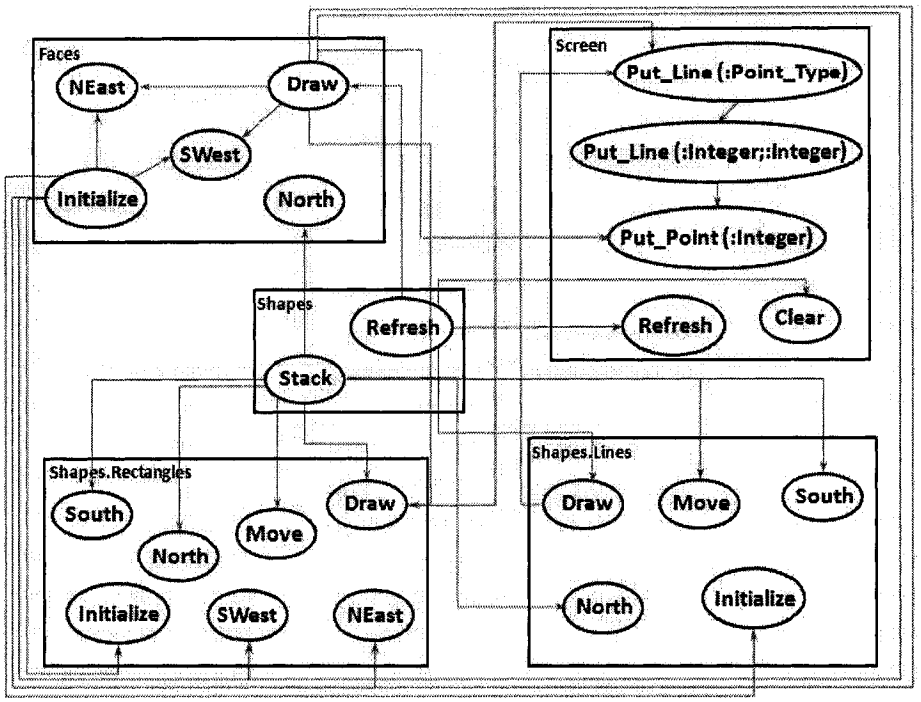

Figure 13: The call graph of project Shapes.

visualize them as a call graph as shown in Figure 13[1]. In this call graph E, R and M are as follows:

$$E = \{\ Refresh, Clear, Draw(:\ Face\_Type),$$
$$NEast(:\ Face\_Type), NEast(:\ Rectangle_Type),$$
$$Put\_Line(:\ Point\_Type), Put\_Line(:\ Integer;:\ Integer),$$
$$\ldots\ \}$$

$$R = \{\ < Shapes.Refresh, Screen.Clear >,$$
$$< Shapes.Refresh, Faces.Draw(:\ Face\_Type) >,$$
$$< Faces.Draw(:\ Face\_Type), Faces.SWest(:\ Face\_Type) >,$$
$$\ldots\ \}$$

[1]In order to improve clarity of the figure, the signatures of the subprograms are not shown, except in the case of overloaded subprograms.

$$M = \{\ Shapes, Shapes.Rectangles, Shapes.Lines, Faces, Screen\ \}$$

Based on the number of entries in the related tables in the database, $|E| = 24$ as the number of subprograms defined in 5 packages ($|M| = 5$), and the number of call relations between these subprograms to run the scenario are 491 ($|R| = 491$). It is important to mention that we should not include the `main` package and `main` subprogram in measuring the coupling and cohesion since these are some auxiliary components added to database for data consistency and visualization. The corresponding Intra-Package and Inter-Package graphs can be extracted from this call graph, but due to space limitations,they are not shown here.

7.1 Quality measurement of the system

Cohesion Table 7 lists the value and level of cohesion of each package in this project. Based on the result, most of the packages in this system have a low degree of cohesion.

Table 7: Cohesion measure of the packages of project `Shapes`.

Package	Cohesion	n	Cohesion Level
`Shapes`	0	2	Low
`Shapes.Rectangles`	0	7	Low
`Shapes.Lines`	0	5	Low
`Faces`	0.4	5	Medium
`Screen`	0.2	5	Low
`System`	0.12	–	–

Also as it is shown in the call graph, the Intra-Package edges exist only in two of

Table 8: Coupling measure of the packages of project **Shapes**.

Coupling of	Shapes	Shapes. Rectangles	Shapes. Lines	Faces	Screen	System	Coupling Level
Shapes	0	4	4	2	2	12	Greater than threshold
Shapes.Rectangles	0	0	0	0	1	1	Weak
Shapes.Lines	0	0	0	0	1	1	Weak
Faces	0	6	1	0	1	8	Strong
Screen	0	0	0	0	0	0	Weak
System	0	10	5	2	5	4.4	–

the packages in the graph (**Faces** and **Screen**). In other packages, there is no relation among the subprograms inside the package, implying that the package contains the implementation of some different and unrelated concerns. For instance, packages **Shapes.Rectangles** and **Shapes.Lines** contain a set of subprograms which are only called by other packages. These results imply that some refactorings are needed to localize the most related subprograms in one cohesive package. For example, subprogram **Initialize** separated in three packages can be localized into one package that exclusively implements this concern.

Coupling Table 8 lists the value and level of coupling of packages in this project.

Based on the result, package **Shapes** holds the coupling value even greater than the threshold, implying that it is strongly coupled to other packages. That is because it contains two unrelated subprograms, **Refresh** and **Stack**, each of which is sending more than one message to every package. Also package **Faces** is strongly coupled to the functionalities of the other packages. Other packages have a weak level of coupling, since they are sending at most one message to only one package in the system. This shows that they are able to handle a concern with the minimum dependency to other

Table 9: Modularity measure of the packages of project **Shapes**.

Modularity of	Cohesion Level	Coupling Level	Modularity Level	Modularity Value
`Shapes`	Low	Greater than threshold	Low	0
`Shapes.Rectangles`	Low	Weak	Medium	0
`Shapes.Lines`	Low	Weak	Medium	0
`Faces`	Medium	Strong	Medium	0.04
`Screen`	Low	Weak	Medium	0.2
`System`	–	–	–	0.022

subprograms or packages.

Modularity Having the cohesion and coupling of the system packages, we can evaluate the modularity of each package using the metrics proposed in Section 6.3. The modularity of packages in the system is listed in Table 9.

The combination of both value and level of the modularity for a given package can be used to compare or reason about the modularity of the package. For instance, based on the results shown in Table 9, we conclude that package **Shapes** has the least modularity in this system, that is because it has the least possible cohesion which is 0 and a high value even greater than the threshold for coupling. This implies that not only this package has a very low degree of quality, but also it has degraded the overall modularity of the system. Some reengineering tasks are required to restructure the package, so that its quality gets improved. Even though packages `Shapes.Rectangles` and `Shapes.Lines` hold a weak level of coupling, they are not very cohesive. However, they have better quality than package **Shapes**. Package `Faces` is the most cohesive package, but it is highly coupled to other packages. Package `Screen` is not coupled to other packages, but it has a low level of cohesion. Overall, packages `Screen` and `Faces`, tend to have higher quality among all packages in the system.

7.2 Quality measurement of the refactored system

The goal of this section is to validate the proposed metrics for cohesion, coupling and modularity of the packages in a system. Essentially, obtaining measurements on a system is analogous to taking a snapshot of the status of its quality in terms of a set of quality attributes which are of interest to the analysis at the time. By implementing refactoring, we expect the quality of the system to improve for the same set of quality attributes. In this case study we take two snapshots, one before refactoring and one after refactoring. We then we compare the two snapshots to determine the degree to which refactoring has improved the quality attributes under consideration. To evaluate the system we run it under the same scenarios and extract the execution traces. Then, we form the call graph to observe the dependencies in the refactored system. In the next step, we apply the metrics on the system to measure the cohesion, coupling and modularity of the packages. Finally, we compare the obtained measures from the original and refactored system to validate whether or not the proposed metrics provide measures which are consistent to our expectations.

How to determine which refactoring strategies are the most suitable for this system is out of the scope of this research. However, based on the exhibited anomalies by the packages and the guidelines discussed in [23], we chose the following changes to apply to the system:

- Localizing the initialization concern in one newly created package, named `Initialization` and moving three subprograms `Initialize` defined in three packages

`Shapes.Rectangle`, `Shapes.Lines` and `Faces` in package `Initialization`. We expect that this task will reduce the coupling between the three packages `Shapes.Rectangle`, `Shapes.Lines` and `Faces` and builds a new cohesive package.

- Transferring subprogram `Refresh` from package `Shapes` to package `Screen`. The reason is that the functionality of subprogram `Refresh` is clearing and refreshing the screen and drawing some shapes. Thus, it is more related to screen related concerns than shape concerns. By applying this change, we expect to improve the measures for coupling and cohesion of package `Shapes`.

- Applying the polymorphism feature in packages `Shapes`, `Shapes.Rectangle`, `Shapes.Lines` and `Faces` by defining a subprogram `Stack` in both `Shapes.Lines` and `Shapes.Rectangle` packages. This way, when the subprogram `Stack` in package `Shapes` is called, based on the received argument, this subprogram delegates the call to its corresponding child, either `Shapes.Rectangle`, `Shapes.Lines` or `Faces`. By applying this change, we expect a decrease of coupling of package `Shapes`, and an increase of cohesion of its children packages.

We apply the changes, run the program, and make sure that the behavior of the system is preserved (one of the criteria of the refactoring concept). Due to space limitations, we do not show the extracted execution traces, and Intra-Package and Inter-Package graphs and only the obtained call graph is shown (Figure 14). In the following subsections, cohesion, coupling and the modularity of the packages are measured.

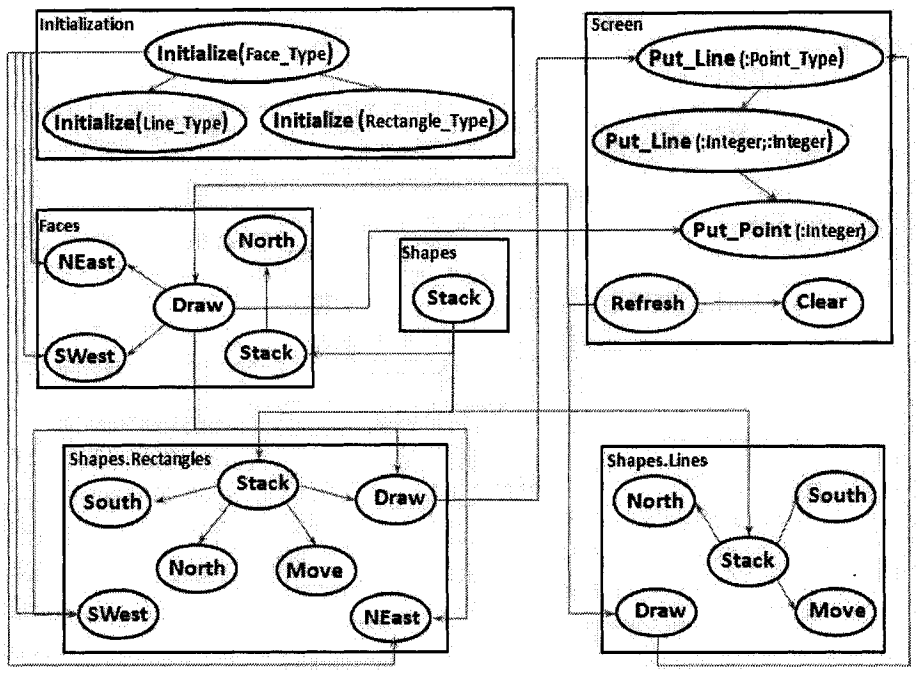

Figure 14: The call graph of the refactored project **Shapes**.

Table 10: Cohesion measure of the packages of the refactored project **Shapes**.

Package	Cohesion	n	Cohesion Level
Shapes	1	1	High
Shapes.Rectangles	0.2	7	Low
Shapes.Lines	0.3	5	Low
Faces	0.3	5	Low
Screen	0.3	5	Low
Initialization	0.66	3	High
System	0.46	–	–

Cohesion Table 10 lists the value and level of cohesion of each package in the refactored system.

The comparison of the cohesion values of packages and system in the original and refactored system are shown in Figure 15. Based on the obtained results, the cohesion of all packages, except package **Faces**, has increased particularly for package **Shapes**, also the cohesion of the system has increased significantly from 0.12 to 0.46. We observe that the obtained results are based on our expectations from the changes.

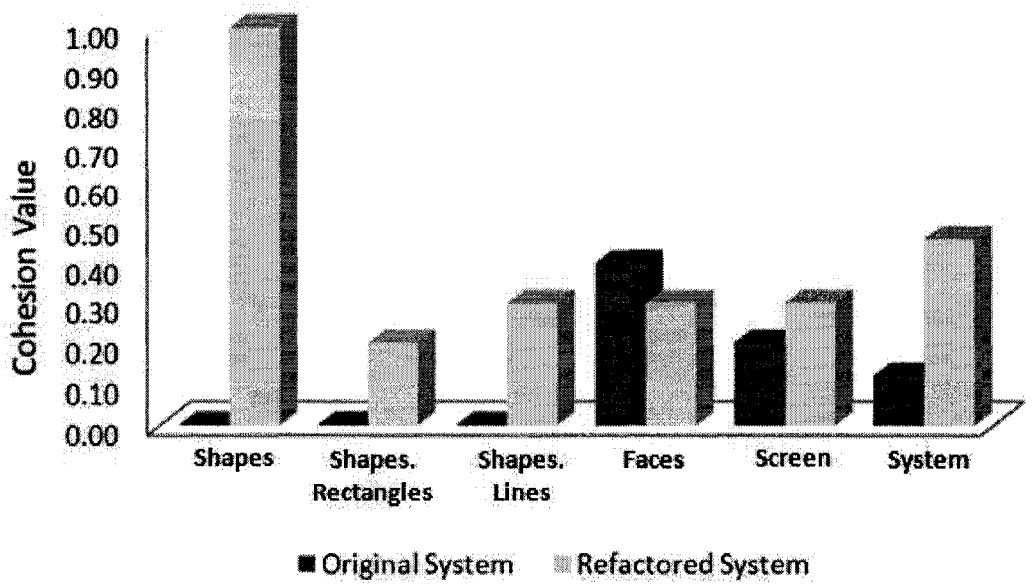

Figure 15: Comparison of the cohesion values of the original and the refactored systems.

This observation validates the applicability and correctness of the proposed metric for measuring the cohesion of the system and its packages.

Coupling Table 11 lists the value and level of coupling of packages in this project. The comparison of the coupling values of packages and system in the original and refactored system are shown in Figure 16. Based on the obtained results, the coupling of packages `Shapes` and `Faces` has decreased, while the coupling of packages `Shapes.Rectangles` and `Shapes.Lines` has not changed since they had a weak level of coupling in the original system. The only package whose coupling has increased is package `Screen`. However, the overall coupling value of the system has decreased noticeably from 4.4 to 2.5. We observe that the obtained results are based on our expectations from the refactorings. This observation validates the applicability and correctness of the proposed metric for measuring the coupling of the system and its

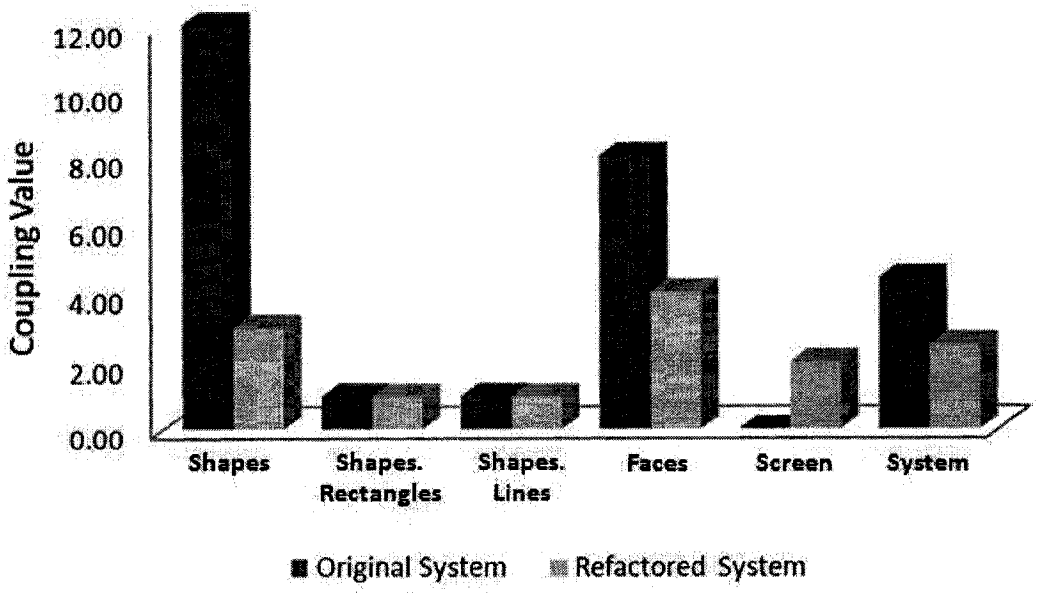

Figure 16: Comparison of the coupling values of the original and the refactored systems.

packages.

Table 11: Coupling measure of the packages of the refactored project **Shapes**.

Coupling of	Shapes	Shapes. Rectangles	Shapes. Lines	Faces	Screen	Initial-ization	System	Coupling Level
Shapes	0	1	1	1	0	3	0	Weak
Shapes.Rectangles	0	0	0	0	1	0	1	Weak
Shapes.Lines	0	0	0	0	1	0	1	Weak
Faces	0	3	0	0	1	0	4	Weak
Screen	0	0	1	1	0	0	2	Weak
Initialization	0	2	0	2	0	0	2	Weak
System	0	6	2	4	3	0	2.5	–

Modularity The modularity of packages in the system is listed in Table 12.

The comparison of the modularity values of packages and system in the original and refactored system are shown in Figure 17. Based on the obtained results, the modularity of all packages, except package **Screen** has increased. Modularity of package **Screen** has decreased due to the increase of its coupling from 0 to 2. However,

Table 12: Modularity measure of the packages of the refactored project **Shapes**.

Modularity of	Cohesion Level	Coupling Level	Modularity Level	Modularity Value
`Shapes`	High	Weak	High	0.25
`Shapes.Rectangles`	Low	Weak	Medium	0.1
`Shapes.Lines`	Low	Weak	Medium	0.15
`Faces`	Low	Weak	Medium	0.06
`Screen`	Low	Weak	Medium	0.1
`Initialization`	High	Weak	High	0.22
`System`	–	–	–	0.13

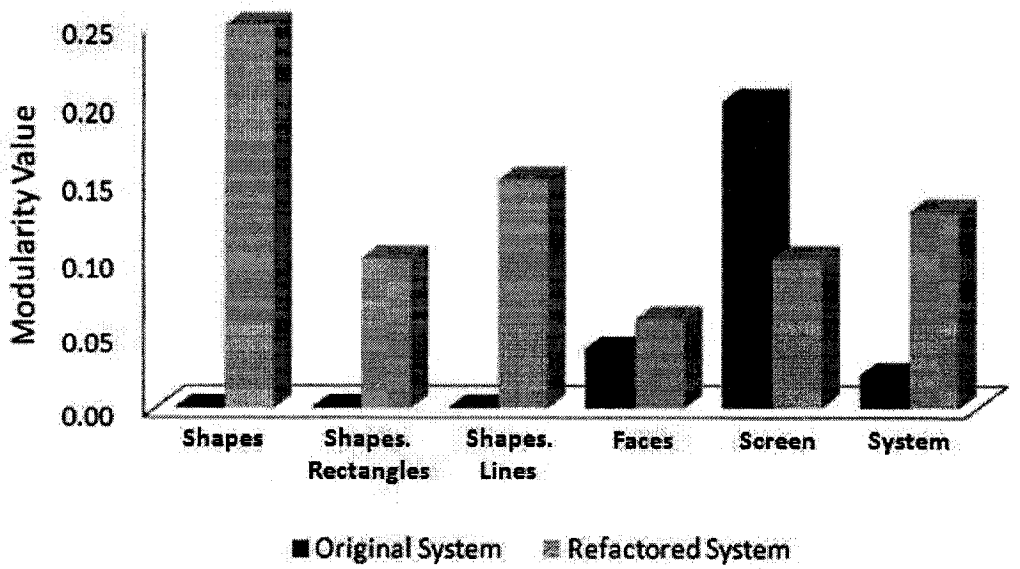

Figure 17: Comparison of the modularity values of the original and the refactored systems.

the overall modularity value of the system has increased significantly from 0.022 to 0.13. We observe that the obtained results are based on our expectations from the refactorings. This observation validates the applicability and correctness of the proposed metric for measuring the modularity of the system and its packages.

As a conclusion of this chapter, we observe that the obtained results of this system before and after verify that the proposed metrics are appropriate indicators of system internal attributes as cohesion, coupling and modularity. Having these indicators,

maintainers of the system are able to find potential packages that tend to decrease the overall quality of the system. These metrics also can be used as indicators to reveal whether or not a potential change to the system would degrade the overall quality of the system.

Chapter 8

Tool support

In this chapter, we describe the details of the underlying mechanisms used for dynamic analysis tools in the following paragraphs.

In order to support the proposed methodology in this research, we have developed a prototypical tool, named *ADynA: Ada Dynamic Analyzer*. This tool is developed to analyze the gathered raw data on dynamic events during runtime. We expect that the task of code instrumentation is completed (with AspectAda or otherwise) and the traces are obtained and stored. The input to the tool is the text file containing all calls made at runtime. Figure 18 shows the first tab of the tool that the user can choose the input traces file through *Load File* button. After the file is selected, all recorded calls get loaded into the call view and are ready to get analyzed by pushing *Analyze File* button.

8.1 ADynA features

The result of analysis functionality of the tool can be categorized as follows:

- System comprehension through the extraction of information about both structure of the system components and also dynamic behavior of the system during execution. The result of structural analysis, shown in *Program Components* tree view of Figure 19, is a tree view of the packages and their subprograms witch are involved in the execution of that specific scenario. The result of behavioral analysis, shown in *Messages* list of Figure 19, is a list of all messages passed between the above mentioned packages and subprograms during the execution of that specific scenario. The messages in this list are ordered based on the time they have been generated.

- Graph-based visualization to provide a better representation of the system dynamic behavior. The call graph extracted from the message passing list is generated and drawn in the *Graph Visualization* view of the tool, shown in Figure 20.

- Measurement analysis through applying the metrics mentioned in Chapter 6 on the list of messages. The corresponding tables, shown in Figure 21, represent the values and levels of cohesion, coupling and modularity of each package during the program execution.

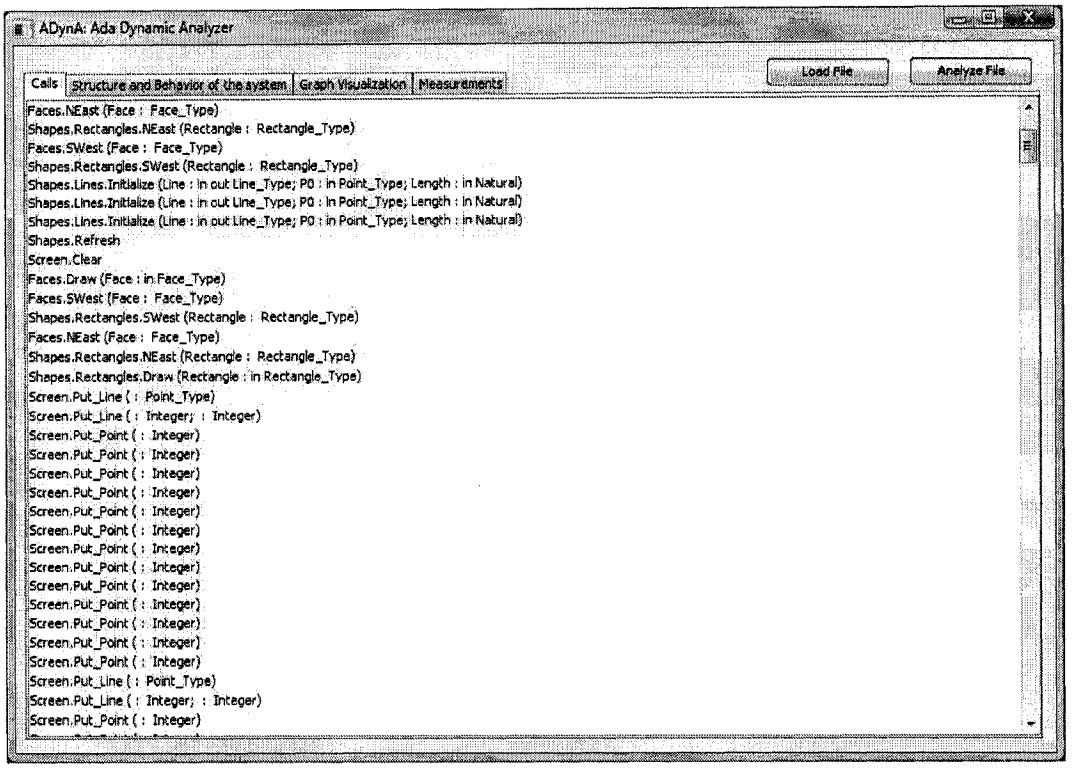

Figure 18: ADynA: Ada Dynamic Analyzer, Calls view.

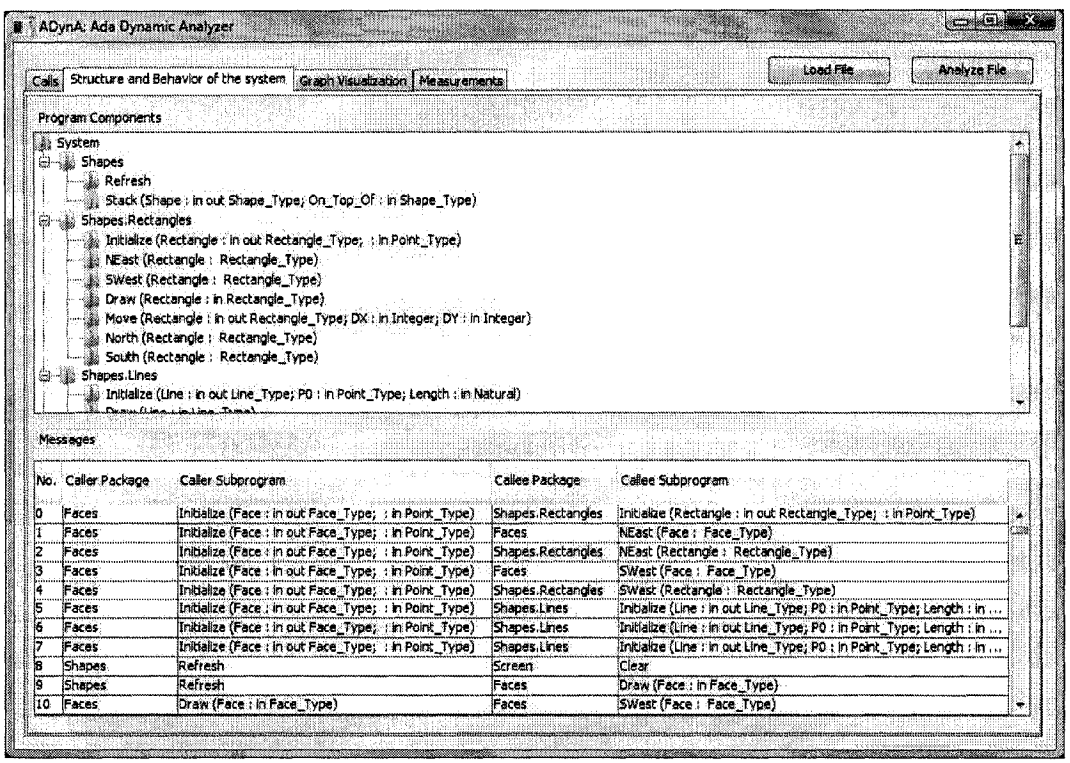

Figure 19: ADynA: Ada Dynamic Analyzer, Structure and Behavior view.

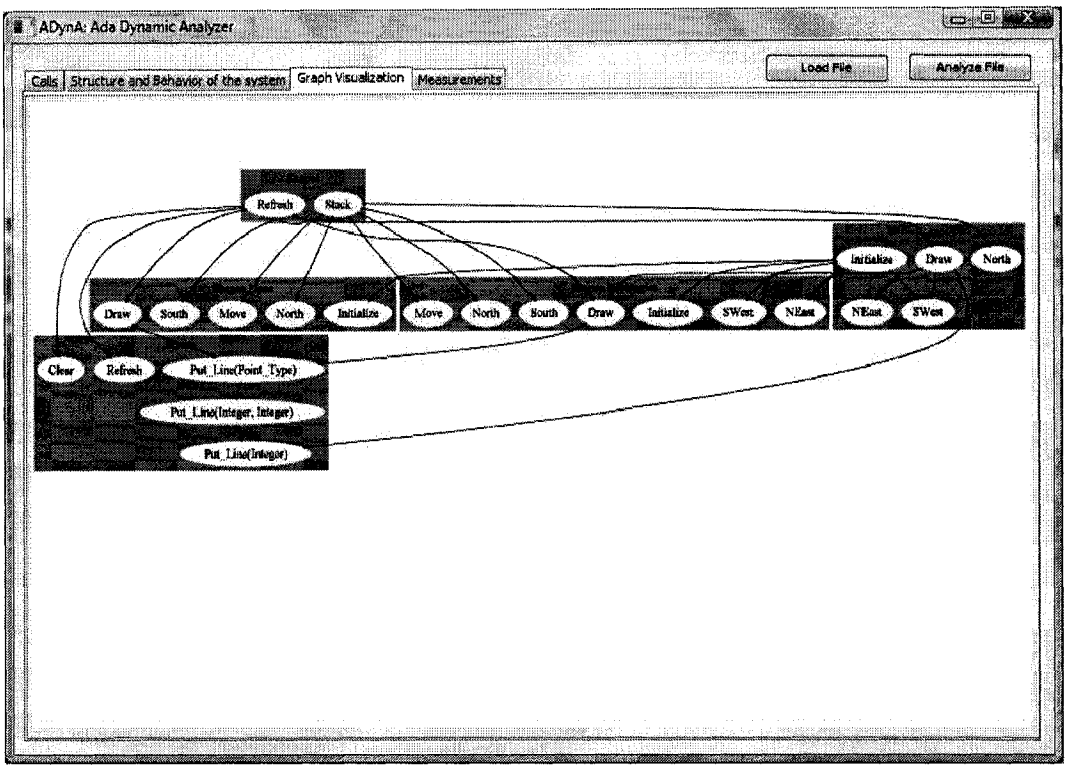

Figure 20: ADynA: Ada Dynamic Analyzer, Graph Visualization view.

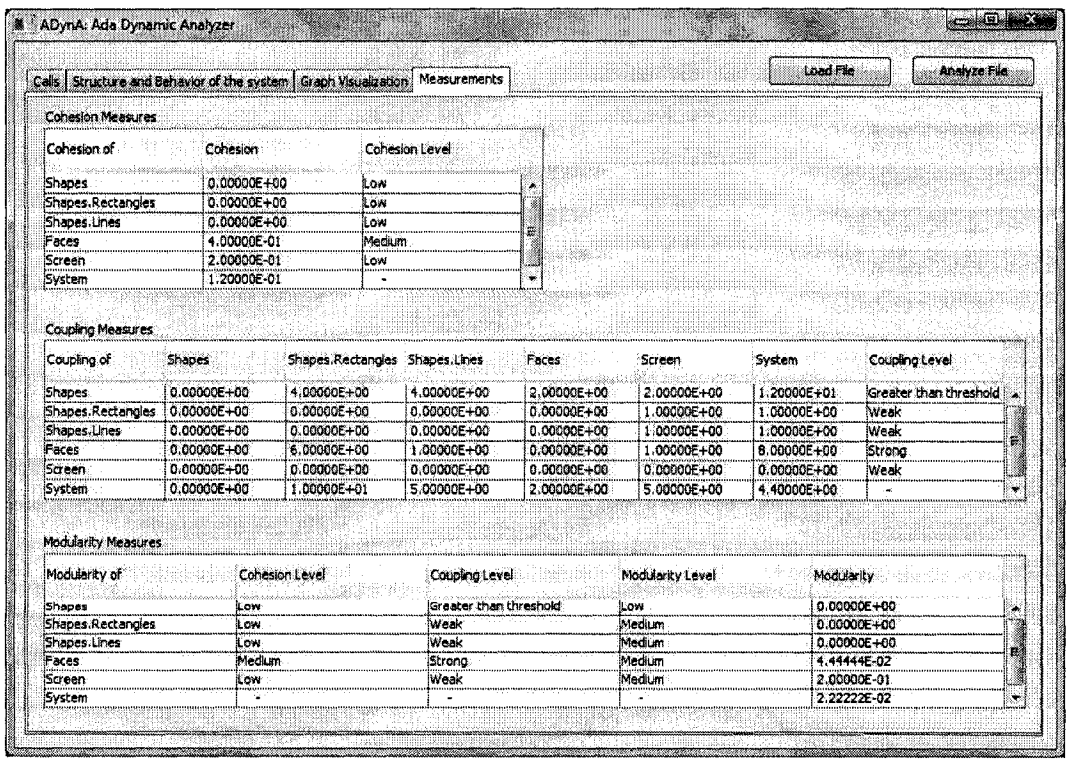

Figure 21: ADynA: Ada Dynamic Analyzer, Measurements view.

8.2 ADynA implementation

ADynA is implemented in Ada, however only for Graphical User Interface (GUI) we have used Java, since Ada does not fully support graphical representations. Also for call graph representation, we utilized an stand-alone tool named Graphvis 2.20 (Graph Visualization Software). Graphvis is an open-source graph visualization software that takes descriptions of graphs in a simple text language, and make diagrams in several formats including call graph. MySQL is used as the Relational Database Management System (RDMS) in ADynA for data storage and retrieval.

8.3 ADynA adaptability

One of the objectives of ADynaA design and implementation is to deploy an adaptable tool which is capable to analyze a set of object-oriented program including Ada, Java, C++, C# and etc. The only required condition for including a programming language in this set is that the language should support the concept of the units of modularization (package) each of which including a set of functionalities (subprogram). Any functionality in a unit should be able to communicate with (call) accessible functionalities inside the same surrounding unit, and also any unit should be able to communicate with (call) the accessible functionalities included inside other units. Once the list of calls (with a specific format mentioned in Section 5.3) are generated by an instrumentation tool, ADynA can store the traces in the database and through the same queries mentioned in Sections 6.1, 6.2, 6.3, the values and levels of cohesion, coupling and modularity of the units or the system can be evaluated.

8.4 ADynA limitation

One limitation that we can consider for ADynA is the scalability of the call graph. The call graph generated from the execution of a large-scale program has a lot of nodes and edges. However the tool used for graph visualization (Graphvis) does not allow to zoom. As the result the call graph will become difficult to understand and trace. Adding some levels of abstraction for visualization can help in categorizing a set of subprograms or packages inside a abstract unit. This way, we can visualize a more clear graph.

Chapter 9

Related work and evaluation

The proposed approaches for system analysis are mostly specialized for some specific needs. In this field, proposed metrics for measuring and evaluating the system internal quality attributes such as cohesion, coupling are the basic proposals in the literature which are either general for any object-oriented system or defined for a specific language. We discuss the language-independent proposals in the next two sections 9.1 and 9.2, following by Ada specific analysis approaches in Section 9.3.

9.1 Cohesion

Several approaches have been proposed in the literature to measure cohesion in an object-oriented system each of which falling under different categories based on the kind of cohesion they measure.

9.1.1 Measuring structural cohesion

This is the most investigated category and the one followed by our approach. The main idea behind measuring structural cohesion is to investigate the level of relations between class (package) elements, such as variables and methods (subprograms). Subprograms are related when they either use the same set of variables or invoke each other. Some commonly used and known metrics defined in this category are as follows:

LCOM1 [14]: Chidamber and Kemerer have defined LCOM1 (Lack Of Cohesion) for measuring the class cohesion as the degree to which the methods are similar in accessing the instance variables. LCOM1 is the number of pairs of methods that share no instance variables.

LCOM2 [13]: In the later work, Chidamber and Kemerer have proposed LCOM2 as subtraction of the number of pairs of methods with shared instance variables from the number of pairs of methods without shared instance variables. If the result is negative, the cohesion measure will be 0.

LCOM3 [43]: Li and Henry have defined this measure as follows: Consider an undirected graph G where the vertices are the methods of a class and there is an edge between two vertices if the corresponding methods share at least one instance variable. LCOM3 is defined as the number of connected components in graph G.

LCOM4 [34]: Hitz and Montazeri have added the concept of call relation to the cohesion measure defined by Chidamber and Kemerer and restated by Li and Henry. LCOM4 is the same as LCOM3 where graph G additionally has an edge between vertices representing methods a and b, if method a invokes method b or vice versa.

C [33]: Hitz and Montazeri have defined C (Connectivity) which discriminates classes having LCOM4= 1 by taking into account the number of edges of the connected component. C is a linear mapping of the interval [n-1, n.(n-1)/2] (as the result of LCOM4) onto the interval [0, 1] as follows: Let V be the vertices of graph G from LCOM4 and E its edges, then

$$\mathrm{C} = 2\frac{|E|-(n-1)}{(n-1)(n-2)}$$

LCOM5 [33]: LCOM5 is defined as follows: Consider a set of methods M_i $(i = 1, ..., m)$ accessing a set of instance variables A_j $(j = 1, ..., a)$. Let $\mu(A_j)$ be the number of methods that reference A_j. Then

$$\mathrm{LCOM5} = \frac{(\frac{1}{a})\sum_{j=1}^{a}\mu(A_j)-m}{1-m}$$

TCC and LCC [5]: The measure TCC (Tight Class Cohesion) is the relative number of directly connected methods and LCC (Loose Class Cohesion) is the relative number of directly or indirectly connected methods. They are defined as follows: Let NP be the maximum number of public method pairs, that is, $NP = \frac{N*(N-1)}{2}$ for N public methods. Let NDC be the number of direct connections and NIC be the number of indirect connections between public methods. Then TCC is defined as $TCC = \frac{NDC}{NP}$ and LCC as $LCC = \frac{NDC+NIC}{NP}$.

RCI [7]: RCI has been introduced for measuring the cohesion of a class in object-oriented systems as a ratio of number of all existing interaction between methods and data declaration in a class to the number of all possible interactions.

CBMC [11]: CBMC has been defined for measuring cohesion of a class. Chae et al. have used reference graph representing the relations among methods and instance

variables in a class. They believe that two factors affect the cohesion of a class. First, the connectivity of the reference graph and second, the pattern of interactions among the constitute components of a class.

ICBMC [69]: Zhou et al. have analyzed the limitations with the existing measures for cohesion and finally they have proposed ICBMC as an improved measure for cohesion based on CBMC defined by Chae in [11].

DRC [68]: Zhou at al. have defined DRC as a more precise measure for cohesion of a class than those proposed before. DRC uses a class member dependence graph to represent all kind of dependencies among class members. Cohesion is measured based on four types of basic dependency as: 1) Read dependence from methods to attributes, 2) Write dependence from attributes to methods, 3) Call dependence between methods and 4) Flow dependences between attributes. They believe that considering four above mentioned types of relations can better characterize all relations among the members of a class.

DMC [63]: The definition of DMC is similar to the definition of DRC measure. The authors have proposed the construction of an adjacency matrix to be able to reflect the dependence degree of explicit dependencies among the nodes in the dependence graph. Comparing with the previous cohesion measures, they believe that in the previous works, neither the direction of dependencies between methods and attributes, nor flow dependencies and potential dependencies are characterized. DMC precisely considers the relationships among the elements in a class, which characterizes not only four types of explicit dependencies (read dependence, write dependence, call dependence, and flow dependence), but also the implied indirect and potential dependencies.

In addition, Briand [7] has defined a unified framework that classifies and compares all the above mentioned metrics for cohesion. Data flow between the methods is the principle factor for measuring cohesion. Most of the above mentioned measures consider only instance variables used or shared between methods for calculating the class cohesion. The only metrics that consider call relations (method invocation) among the methods as well as instance variable usage are LCOM4 and DRC measures.

9.1.2 Measuring conceptual cohesion

Another category of metrics are proposed to measure how class elements are logically related. The main idea behind this category relies on the fact that a cohesive class is a unit whose elements are conceptually and functionally strongly related. The methodology behind this class of cohesion is based on analysis of the semantic information embedded in the source code, such as comments and identifiers. This category includes metrics such as LORM (Logical Relatedness of Methods) [19], C3 (Conceptual Cohesion of Classes) [45] which is based on semantic information in the source code and the composite cohesion metric proposed by Patel et al. [50] that measures the information strength of a module.

9.1.3 Measuring other types of cohesion

Some other cohesion metrics have been proposed based on other aspects of the system such as Slice-based metrics [47] as a complementary views of cohesion to the structural metrics, cohesion for knowledge-based systems [39], dynamic cohesion metrics

for distributed applications [15], information theory-based metrics [2] and metrics proposed for measuring cohesion in aspect-oriented programs [26, 67].

9.2 Coupling

Several approaches have been proposed in the literature to measure coupling of object-oriented systems. They can be categorized based on the kind of coupling they measure.

9.2.1 Measuring structural coupling

This is the most investigated category in the literature and the one followed by our approach. Here, coupling is based on method invocations and attribute references. Classes are coupled when they call methods of each other or refer to attributes of each other. Some known metrics defined in this category are as follows:

MPC [43]: Li and Henry have proposed MPC (Message Passing Coupling) for measuring the coupling of a class. MPC of class i is defined as the numbers of sent messages which are defined in class i.

DAC [43]: Another measure proposed by Li and Henry is DAC (Data Abstraction Coupling) which is defined as the number of abstract data types (ADTs) defined in a class as the class attributes. This measure indicates the number data structures dependent on the definition of other classes. This measure does not consider message passing or call relation among classes.

RFC [14] : The Response For a Class (RFC) is defined by Chidamber and Kemerer

as a set of methods that can be potentially executed in response to a message received by an object of that class. As explicitly mentioned, this measure is concerned about the potential calls, not guaranteed calls at runtime.

RFC_α [8]: In the definition of RFC by Chidamber and Kemerer, RFC does not only include the methods directly invoked by a method, but also the methods called by these methods and so on, implying that the levels nested method invocations are not considered in this definition. Briand et al. [8] have added the levels of message passing to RFC and defined RFC_α, where $\alpha = 1, 2, 3, ...$ Then, they have considered the RFC measure proposed by Chidamber and Kemerer as special case of RFC_α, where:

$$RFC(c) = RFC_1(c)$$

CBO_1 [14]: Chidamber and Kemerer have defined CBO (Coupling Between Objects) for measuring coupling of a class. It is defined as the count of non-inheritance related couples with other classes. Two objects are considered as coupled if they act upon one another, implying that an object of one class uses the methods or instance variables of the other. This measure is calculated based on static information of the source code.

CBO [13]: In the later work, Chidamber and Kemerer have revised the definition of CBO_1. In the new definition, CBO for a class is the number of other classes to which this class is coupled. Similar to CBO_1, static analysis is used for calculating CBO.

ICP [42]: ICP (Information-flow-based Coupling) is defined by Lee and Liang as

the number of polymorphistically invoked methods of other classes, weighted by the number of parameters of the invoked method.

A suite of coupling measures (IFCAIC, ACAIC, OCAIC, FCAEC, DCAEC, OCAEC, IFCMIC, ACMIC, DCMIC, FCMEC, DCMEC, OCMEC, IFMMIC, AMMIC, OMMIC, FMMEC, DMMEC, OMMEC) [6] is defined by Briand et al. where the authors distinguish between different type of relationship between classes such as friendship, inheritance and none, different types of interactions such as class-attribute, class-method and method-method interactions, and the locus of the interaction impact.

- The first or first two character of the name of each measure refers to the type of coupling relationship between classes as follows: A: Ancestor, D: Descendant, F: Friend classes, IF: Inverse Friends (classes that declare a given class a as their friend) and O: Others.

- The next two characters indicate the type of interaction as follows: CA: class-attribute interaction from class a to class b, implying that an attribute of class a is of type class b, CM: class-method interaction from class a to class b, if a newly defined method of class a has a parameter of type class b and MM: method-method interaction from class a to class b, if a method implemented in class a statically invokes a method of class b, or receives a pointer to such a method.

- The last two characters shows the locus of the interaction impact as follows: IC: Import coupling, counts the number of other classes called by the class and EC: Export coupling, count number of other classes using class x.

However, the authors assign no weight to these different kinds of interactions.

Also a framework is proposed by Hitz and Montazeri [34] which distinguishes between object level and class level coupling.

In all above mentioned measures are static coupling measures, since the class relations are obtained through static information of the source code. However, the empirical studies indicate that static analysis of the code can not provide a complete set of information about the system dependencies due to polymorphism and inheritance principles of object-oriented programming. Arisholm [3] has introduced the concept of dynamic coupling measure as a complementary approach to static coupling measures. They measure the coupling based on actual call relations at runtime of the program. A tool is implemented to collect the dynamic data at runtime and apply the defined measures. Their results have indicated that dynamic coupling measures can capture different properties than static coupling measures.

DCM [32]: Hassoun et al. have defined DCM (Dynamic Coupling Metric) in reflective systems as a metric for measuring system coupling based on dynamic analysis of message passing and method invocation. DCM of an object P during a time period Δt, is defined as the sum of all program execution steps and the sum of total number of objects which are coupled to object P. If the program consist of one object, then the coupling measure of the system is zero.

In addition, Briand [8] has defined a unified framework that classifies and compares all the above mentioned metrics for coupling.

9.2.2 Measuring conceptual coupling

Conceptual coupling is the degree to which the classes are logically dependent on each other. It uses the concept of semantic similarity between elements of the source code, such as comments and identifiers. Some work has been done to measure conceptual coupling such as [53] by Poshyvanyk and Marcus, also the research in software clustering field [44] use the same concept as semantic similarities in the source code.

9.2.3 Measuring other types of coupling

Several metrics are proposed in the literature to measure coupling of different types of software systems, such as coupling metrics for knowledge-based systems [39], coupling metrics for aspect-oriented programs [66] and information theory-based metrics [2].

9.3 Comprehension and measurement in Ada

Most of the researches about analysis of Ada programs are based on static analysis. They mainly focus on the structural dependencies in the code to obtain information about fault-prone components, early error detection and metrics. Examples of such works are as follows:

In the field of comprehension of an Ada program, Raza et al. have designed and implemented, Bauhaus [57], a comprehensive tool suite that supports program comprehension and reverse engineering on all layers of abstraction, from source code to architecture. The tool is implemented in Ada and it is capable to analyze programs written in Ada, C, C++ and Java. The tool utilizes both dynamic and static

approaches to do control-flow and data-flow graphs analysis, pointer analysis, side effect analysis, program slicing, clone recognition, source code metrics, static tracing, query techniques, source code navigation and visualization, object recovery, remodularization and architecture recovery techniques. The metrics implemented in Bauhaus operate on different levels of a software system, source code level and architecture level. On the source code level, metrics such as lines of code, Halstead, maximal nesting and cyclomatic complexity are implemented. On the architecture level, the tool provides metrics such as number of methods, classes and units, coupling and cohesion, and derived metrics such as number of methods per class and classes per unit. However, the measures used for deriving the metrics are not mentioned in the paper. The calculated results of the metrics can be used to estimate software complexity, to detect code smell or to provide parameters to maintenance effort models. The objective of this work is close to our research, but the result of their work is a general tool suite for comprehension while in our research. We are mostly concerned about measurement of system modularity through measuring coupling and cohesion.

In the field of program analysis, Laski et al. in [41] have performed dependency analysis of Ada programs. Discussing some limitations of the result of existing static analysis approaches such as undecidability, inability to offer an explanation of the reported events and its inadequacy to handle events that occur on individual program paths, they have a modified dependency model as a solution which also can be beneficial in dynamic analysis of Ada programs. Their modifications include partial vs. total definitions of Ada arrays, new concept of reaching definitions, potential vs. guaranteed dependencies, inter-procedural dependencies, and an explanation feature.

They also investigate path analysis as a novel method for the identification of dependencies along individual program paths. Based on their proposal, the modified dependency model can be beneficial in integrating dynamic and static analysis in order to analyze Ada programs. Their main goal is to define a more accurate model for procedure calls, suitable for catching undetectable data flow anomalies and a better solution for analysis of error creation and propagation in testing and debugging. Comparing the objectives of their work with our research, we can observe that the objectives behind each work is to analyze Ada programs to obtain some information about the existing anomalies in the code, however their methodology is to define a new form of dependency model, while we extract the anomalies through measurement of coupling and cohesion of system components.

Pritchett in [55, 56], has defined a set of metrics for Ada 95 in order to find fault prone packages in the code. The proposed measures are based on those defined by Chidamber and Kemerer in [13]. The factors measured in this work are depth of inheritance, total number of children, total attributes, local attributes, total operations, local operations, overridden operations, class-wide operations, message passing coupling, attributes of abstract data types, abstract data types referenced, class cohesion and max cyclomatic complexity. AdaSTAT [1] static analysis tool is utilized to collect metrics data for each package. Once collected, the data is entered into the Minitab [48] statistical software package and descriptive statistics for each of the metrics are collected. The main goal of this research is to reason about how each metric affects the fault proneness of the code. For this reason, they have conducted an empirical study. Using both regression analysis and correlation coefficient, they have concluded

that all above mentioned factors, except message passing coupling, class cohesion, overridden operations and class-wide operations, are predictors of fault-prone classes. They have not proposed a specific formula for measuring these factors, and they are calculated based on those proposed traditionally by Chidamber et al. in [13]. As mentioned above, this work is based on the static properties of the code.

In the field of measurement, Cherniack et al. [12] designed a tool for measuring coupling and cohesion of Ada programs. They have used Verdix's DIANA Interface Package (DIP) instead of Ada compiler, since the compiler can not provide the needed data for calculating the coupling and cohesion of an Ada program. The tool is based on traversing Descriptive Intermediate Attributed Notation for Ada (DIANA) nets, that extracts those Ada program characteristics needed for coupling and cohesion calculations. Even though the metric used in this work is not explicitly mentioned, they believe that cohesion is an intra-module property that measures the interrelationship between the inputs and outputs of calculations and coupling is an inter-module property that measures the interdependence between software modules.

Xu et al. in [64] have proposed an approach to measure the cohesion of a package in an Ada program based on dependence analysis. In their approach, cohesion is separated into three categories of relation: inter-object, subprogram-object and inter-subprogram. They have defined a measure for each category. The calculated results of each measure are integrated to measure the package cohesion independently and also for measuring the cohesion of the system as a whole. In their work, the methodology for measuring the inter-subprogram cohesion is similar to the measure defined in our work. Based on their measures, the value of cohesion is 1 (maximum) when execution

of each subprogram depends on all other subprograms in the package. Inversely, the value of cohesion is 0 (minimum) when all subprograms have no relation with any other subprograms in the same package. The difference of their approach with the approach defined in this research is that they have used static analysis to extract the dependencies among subprograms and objects, which can not be sufficient in case of main object-oriented principles as polymorphism and dynamic bindings. They have mentioned this task as their future work.

To summarize the related work to this research, we can conclude that all analysis approaches mentioned above, except Bauhaus [57], are based on static analysis. Static analysis alone is insufficient in evaluating the dynamic behavior of an application at runtime, since in object-oriented systems due to dynamic binding, inheritance and polymorphism, dynamic information about the events at runtime of the system can provide a set of valuable and precise information about the system. In this research, we have used dynamic analysis to obtain a set of data about system interaction in the execution time of the program. After analyzing this set of data by applying the measurements and metrics, we detect the problematic packages which need to get refactored or restructured to expose better object-oriented design principles.

Chapter 10

Conclusion and recommendations

10.1 Summary and conclusion

In this research, we proposed an approach for obtaining comprehension of Ada programs. Our approach is based on applying dynamic analysis on Ada program through obtaining execution traces, captured by AspectAda. We refined certain existing metrics for measuring cohesion and also introduced a new set of metrics for measuring coupling and modularity in order to assess the modularity level of the system. This way, we provide an indication of packages whose modularity as an internal quality factor is been degraded due to adding requirements or any other perfective or corrective activity during the evolution and maintenance phase. We believe that this approach can aid maintainers to find potential packages that tend to decrease the overall quality of the system. Our approach is currently supported by automation and a prototypical tool. Our proposed methodology complements the existing approaches for Ada comprehension and measurement by adding the concept of dynamic

analysis, since most of the existing approaches are based on static analysis and are not able to extract dynamic information at runtime such as dynamic binding and polymorphism. Also, the existing methodologies for comprehension of Ada programs are general and can not provide the required set of information for measuring cohesion and coupling. Moreover, the existing metrics in Ada are not sufficient for reasoning about the system modularity. Based on our proposal, defined thresholds for system cohesion, coupling and modularity can help the system maintainers to assess the system quality in terms of system modularity.

10.2 Recommendations

For future work, we intend to extend and import *ADynA* into the *GNAT* package. Also, we plan to extend our measures to apply on aspect-oriented Ada programs. A complementary investigation is to define a set of measures for concurrent program, to check how the coupling, cohesion and modularity of different threads in a running program can be evaluated. The result of this investigation helps to catch the deadlocks of the program, since the deadlock is generally a point of execution where the execution of two or more threads depend on each other.

In order to improve the system modularity, we need to investigate Ada/AspectAda specific refactoring strategies. Also, we intend to work on aspect-oriented migration as a recommended approach to improve the modularity of the system. These are a set of directions related to this research that one can follow as the future work on this research.

Bibliography

[1] Ada Static Analysis Tool. Website http://www.adastat.com/.

[2] Edward B. Allen and Taghi M. Khoshgoftaar. Measuring coupling and cohesion: An information-theory approach. In *Proceedings of the 6th IEEE International Software Metrics Symposium (METRICS)*, pages 119–127, Boca Raton, Florida, USA, November 1999.

[3] Erik Arisholm. Dynamic coupling measures for object-oriented software. In *Proceedings of the 8th International Symposium on Software Metrics (METRICS)*, 2002.

[4] Edward V. Berard. *Essays on object-oriented software engineering.* Prentice-Hall Inc., Upper Saddle River, NJ, USA, 1st edition, 1993.

[5] James M. Bieman and Byung-Kyoo Kang. Cohesion and reuse in an object-oriented system. *SIGSOFT Software Engineering Notes*, 20(SI):259–262, 1995.

[6] Lionel Briand, Prem Devanbu, and Walcelio Melo. An investigation into coupling measures for C++. In *Proceedings of the 19th International Conference on Software Engineering (ICSE)*, pages 412–421, New York, NY, USA, 1997. ACM.

[7] Lionel C. Briand, John W. Daly, and Jurgen K. Wust. A unified framework for cohesion measurement in object-oriented system. In *Proceedings of the 4th IEEE International Software Metrics Symposium (METRICS)*, pages 43–53, 1997.

[8] Lionel C. Briand, John W. Daly, and Jurgen K. Wust. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.

[9] Lionel C. Briand, Sandro Morasca, and Victor R. Basili. Measuring and assessing maintainability at the end of high level design. In *Proceedings of the 9th International Conference on Software Maintenance (ICSM)*, pages 88–97, Washington, DC, USA, 1993. IEEE Computer Society.

[10] Lionel C. Briand, Sandro Morasca, and Victor R. Basili. Property-based software engineering measurement. *IEEE Transactions on Software Engineering*, 22(1):68–86, 1996.

[11] Heung Seok Chae, Yong Rae Kwon, and Doo Hwan Bae. A cohesion measure for object-oriented classes. *Software Practice and Experience*, 30(12):1405–1431, 2000.

[12] Jerome R. Cherniack, Harpal S. Dhama, and Jeanne F. Fandozzi. Tool for computing cohesion and coupling in Ada programs: DIANA dependent part. In *Proceedings of the 12th Ada-Europe International Conference (Ada-Europe)*, pages 180–196, London, UK, 1993. Springer-Verlag.

[13] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[14] Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. *SIGPLAN Not.*, 26(11):197–211, 1991.

[15] E. S. Cho, C. J. Kim, S. D. Kim, and S. Y. Rhew. Static and dynamic metrics for effective object clustering. In *Proceedings of the 5th Asia Pacific Software Engineering Conference (APSEC)*, Washington, DC, USA, 1998. IEEE Computer Society.

[16] Morgan Deters and Ron K. Cytron. Introduction of program instrumentation using aspects. In *Proceedings of the OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, 2001.

[17] Fernando Brito e Abreu and Miguel Goulao. Coupling and cohesion as modularization drivers: Are we being over-persuaded? In *Proceedings of the 5th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 47–57, 2001.

[18] Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *Proceedings of the ICSE Workshop on Dynamic Analysis (WODA)*, 2003.

[19] Letha Etzkorn and Harry Delugach. Towards a semantic metrics suite for object-oriented design. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS)*, Washington, DC, USA, 2000. IEEE Computer Society.

[20] Norman E. Fenton and Martin Neil. Software metrics: roadmap. In *Proceedings of the International Conference on the Future of Software Engineering (ICSE)*, pages 357–370, New York, NY, USA, 2000. ACM.

[21] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach.* McGraw-Hill Inc., New York, NY, USA, 1998.

[22] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Proceedings of the OOPSLA Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, 2000.

[23] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the design of existing code.* Addison-Wesley Longman Publishing Co. Inc., Boston, MA, USA, 1999.

[24] Audun Foyen, Erik Arisholm, and Lionel C. Briand. Dynamic coupling measurement for object-oriented software. *IEEE Transactions Software Engineering*, 30(8):491–506, 2004.

[25] GDB: The GNU Project Debugger. `http://sourceware.org/gdb/`, Last Updated in 2007.

[26] Jean-Franois Glinas, Mourad Badri, and Linda Badri. A cohesion measure for aspects. *Journal of Object Technology (JOT)*, 5(7):75–95, September-October 2006.

[27] GNAT GPL Edition. Downloadable from https://libre.adacore.com.

[28] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: A call graph execution profiler. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.

[29] Thomas Gschwind and Johann Oberleitner. Improving dynamic data analysis with aspect-oriented programming. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR)*, 2003.

[30] Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge. Compression techniques to simplify the analysis of large execution traces. In *Proceedings of the 10th International Workshop on Program Comprehension (IWPC)*, pages 159–168, 2002.

[31] Abdelwahab Hamou-Lhadj, Timothy C. Lethbridge, and Lianjiang Fu. Challenges and requirements for an effective trace exploration tool. In *Proceedings of the 12th IEEE International Workshop on Program Comprehension (IWPC)*, Washington, DC, USA, 2004. IEEE Computer Society.

[32] Youssef Hassoun, Roger Johnson, and Steve Counsell. A dynamic runtime coupling metric for meta-level architectures. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 47–57, 2004.

[33] Henderson-Seller. *Software Metrics.* Prentice Hall, U.K., 1996.

[34] Martin Hitz and Behzad Montazeri. Measuring coupling and cohesion in object-oriented systems. In *Proceedings of the International Symposium on Applied Corporate Computing*, 1995.

[35] ISO International Organization for Standardization. ISO 8402:1994, Quality management and quality assurance, 1994.

[36] ISO International Organization for Standardization. ISO/IEC 9126-1:2001(E), Software engineering - Product quality, Part 1: Quality model, 2001.

[37] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, 2001.

[38] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[39] Stefan Kramer and Hermann Kaindl. Coupling and cohesion metrics for knowledge-based systems using frames and rules. *ACM Transactions on Software Engineering and Methodology*, 13(3):332–358, 2004.

[40] Danny B. Lange and Yuichi Nakamura. Object-oriented program tracing and visualization. *IEEE Computer*, 30(5):63–70, 1997.

[41] Janusz Laski, William Stanley, and Jim Hurst. Dependency analysis of Ada. In *Proceedings of the ACM SIGAda International Conference on Ada (SIGAda)*, pages 263–275, 1998.

[42] Y. S. Lee and B. S. Liang. Measuring the coupling and cohesion of an object-oriented program based on information flow. In *Proceedings of the International Conference on Software Quality (ICSQ)*, pages 47–57, 1995.

[43] Wei Li and Sallie Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111–122, 1993.

[44] Jonathan I. Maletic and Andrian Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23th International Conference on Software Engineering (ICSE)*, pages 103–112, 2001.

[45] Andrian Marcus and Denys Poshyvanyk. The conceptual cohesion of classes. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM)*, pages 133–142, Washington, DC, USA, 2005. IEEE Computer Society.

[46] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall PTR, Upper Saddle RIver, NJ, EUA, 2nd edition, May 1997.

[47] Timothy M. Meyers and David Binkley. Slice-based cohesion metrics and software intervention. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE)*, pages 256–265, Washington, DC, USA, 2004. IEEE Computer Society.

[48] Minitab Software Maintenance Updates. Website http://www.minitab.com/support/maintenance/.

[49] David Lorge Parnas. Software aging. In *Proceedings of the 16th International Conference on Software Engineering (ICSE)*, pages 279–287, 1994.

[50] Sukesh Patel, William Chu, and Rich Baxter. A measure for composite module cohesion. In *Proceedings of the 14th International Conference on Software Engineering (ICSE)*, pages 38–48, New York, NY, USA, 1992. ACM.

[51] Wim De Pauw, David Lorenz, John Vlissides, and Mark Wegman. Execution patterns in object-oriented visualization. In *Proceedings of the 4th Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 219–234. USENIX, 1998.

[52] Knut H. Pedersen and Constantinos Constantinides. Aspectada: Aspect oriented programming for Ada 95. *Ada Letters*, XXV(4):79–92, 2005.

[53] Denys Poshyvanyk and Andrian Marcus. The conceptual coupling metrics for object-oriented systems. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM)*, pages 469–478, 2006.

[54] Roger S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill Inc., New York, NY, USA, 2nd edition, 1986.

[55] William W. Pritchett. Applying object-oriented metrics to Ada 95. *Ada Letters*, XVI(5):48–58, 1996.

[56] William W. Pritchett. An object-oriented metrics suite for Ada 95. *Ada Letters*, XXI(4):117–126, 2001.

[57] Aoun Raza, Gunther Vogel, and Erhard Plödereder. Bauhaus - A tool suite for program analysis and reverse engineering. In *Proceedings of the 11th International Conference on Reliable Software Technologies (Ada-Europe)*, pages 71–82, 2006.

[58] Spencer Rugaber. Program comprehension for reverse engineering. In *Proceedings of the AAAI Workshop on AI and Automated Program Understanding*, 1992.

[59] J. Rumbaugh, I. Jacobson, and G. Booch. The Unified Modeling Language Reference Manual. Addison-Wesley, 1998.

[60] W.P. Stevens, G.J. Myers, and L.L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1997.

[61] Anneliese von Mayrhauser and A. Marie Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, 28(8):44–55, 1995.

[62] Robert J. Walker, Gail C. Murphy, Bjorn N. Freeman-Benson, Darin Wright, Darin Swanson, and Jeremy Isaak. Visualizing dynamic software system information through high-level models. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 271–283, 1998.

[63] Jianmin Wang, Yuming Zhou, Lijie Wen, Yujian Chen, Hongmin Lu, and Baowen Xu. DMC: A more precise cohesion measure for classes. *Information and Software Technology*, 47(3):167–180, March 2005.

[64] Baowen Xu, Zhenqiang Chen, and Jianjun Zhao. Measuring cohesion of packages in Ada 95. *Ada Letters*, XXIV(1):62–67, 2004.

[65] Andreas Zeller. Program analysis: A hierarchy. In *Proceedings of the ICSE Workshop on Dynamic Analysis (WODA)*, pages 6–9, 2003.

[66] Jianjun Zhao. Measuring coupling in aspect-oriented systems. In *Proceedings of the 10th IEEE International Software Metrics Symposium (METRICS)*, 2004.

[67] Jianjun Zhao and Baowen Xu. Measuring aspect cohesion. In *Proceedings of the 7th International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2004.

[68] Yuming Zhou, Lijie Wen, Jianmin Wang, Yujian Chen, Hongmin Lu, and Baowen Xu. DRC: A dependence relationships based cohesion measure for classes. In *Proceedings of the 10th Asia Pacific Software Engineering Conference (APSEC)*, 2003.

[69] Yuming Zhou, Baowen Xu, Jianjun Zhao, and Hongji Yang. ICBMC: An improved cohesion measure for classes. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM)*, 2002.