Integrating Testing Techniques Through Process Programming

(Technical Report 89-18)

Debra Richardson
Stephanie Leif Aha
Leon Osterweil

May 1989

Information and Computer Science
University of California
Irvine, CA 92717

Abstract

Integration of multiple testing techniques is required to demonstrate high quality of software. Technique integration has three basic goals: incremental testing capabilities, extensive error detection, and cost-effective application. We are experimenting with the use of process programming as a mechanism of integrating testing techniques. Having set out to integrate DATA FLOW testing and RELAY, we proposed synergistic use of these techniques to achieve all three goals. We developed a testing process program much as we would develop a software product from requirements through design to implementation and evaluation. We found process programming to be effective for explicitly integrating the techniques and achieving the desired synergism. Used in this way, process programming also mitigates many of the other problems that plague testing in the software development process.

This work was supported in part by the National Science Foundation under grants CCR-8704311 and CCR-8996102, with cooperation from the Defense Advanced Research Projects Agency (ARPA Orders 6100 and 6108, Program Code 7T10).

Note to Reviewers

This paper describes an experiment in integrating testing techniques through process programming. The body of the paper outlines our experimental objectives and the process program development effort. The entire paper is a bit long, but includes as an appendix the process program that we implemented. Although the code is not required to appreciate our experiment, we felt it important to include it as convincing evidence of the value of process programming in this endeavor. We do not believe it is necessary for the reviewer to read all of the code to evaluate the paper effectively.

1 Introduction

The need for high quality software is becoming increasingly acute, and recognition of this need is becoming more widespread. While techniques such as superior development methodologies can do much to improve software quality, there is still a prevailing consensus that software quality can neither be obtained nor convincingly demonstrated without significant testing activities. In recognition of this, software testing researchers and practitioners have developed a spectrum of tools and techniques. As it becomes clear that these tools and techniques have varying strengths and weaknesses, more attention is focussed on integrating tools and techniques.

It is now widely agreed that there is no single, uniform approach to testing and that there is no single, fixed tool or toolset that can be expected to meet the diversity of testing needs. This is not a failing of the tools and techniques, but rather is integral to the nature of the testing activity. While most practitioners agree that testing aims to determine and assure that software products are of "high quality," most testing researchers now agree that there is no absolute, fixed notion of what "software quality" means. Instead there is growing agreement that software must satisfy a variety of qualities such as robustness, functional correctness, efficiency, adaptability, and reliability. Different projects place different emphasis on different qualities and accordingly have different testing requirements. Thus, we believe that software testing should be viewed as a process whose goal is to satisfy testing requirements that must be enunciated as part of the overall software product requirements.

We believe, therefore, that customized testing processes, which integrate multiple techniques, are required to meet specific testing requirements. Effective integration entails more than simply chosing the best testing techniques and coalescing them into a testing system. Effective technique integration must provide more extensive quality assessment and error detection than any single technique, support incremental use of multiple techniques, and yield substantial savings over unintegrated, multiple technique use.

We suggest that testers approach the task of creating customized testing processes to meet individual testing requirements by considering their task to be a software development task. Ideally, software testers should have at their disposal large libraries of reusable software testing techniques and process specifications and designs, along with tool fragments that could be reused and recombined according to the dictates of their requirements. Software testers would then use typical software development techniques to create testing processes. They would design, implement, and execute their processes and finally evaluate them to assure that stated testing requirements are fulfilled, maintaining their testing processes as necessary.

What has just been described is the notion of process programming applied to the testing process. Process programming provides a formalism for defining relationships among tool fragments and software objects in support of technique integration [Ost87]. A process program manages the testing process by integrating testing techniques and manipulating software and test objects generated by the techniques.

Unfortunately, testers currently face a bewildering diversity of techniques and tools, which rarely have well defined specifications and are rarely elegantly decomposed into modules. Thus, current testing tools and techniques are poorly suited to being used to effectively develop customized testing processes. In addition, testers are unfamiliar with how to exploit software product development techniques to develop software processes.

This paper describes a research project aimed at evaluating our hypothesis that existing software testing tools and techniques can be effectively integrated into larger testing processes by decomposing them into smaller modular capabilities and programming these modules into a larger integrated testing process. We attempted to integrate two existing testing techniques — DATA FLOW testing [RW85, CPRZ86] and RELAY [RT88] — into an integrated testing process, which we call DF⋈RELAY (DATA FLOW tie RELAY). We considered the synthesis of the integrated technique to be a software development activity — *i.e.*, we developed a process program combining the two techniques. This paper describes the development activity we engaged in, evaluates our results, and discusses how these results are applicable to the general problem of developing testing process programs.

1.1 Objectives of Testing Technique Integration

Current testing systems offer widely ranging testing capabilities, but provide little or no support for integration and choice of technique application. Testing is already a human intensive effort without having to agonize over integration. Technique integration is vital is to provide effective support for software testers with diverse testing requirements. Integration provides a testing environment that is a cohesive set of testing components and test objects working concertedly.

Appropriate technique integration increases error detection capabilities and provides more extensive quality assessment than any single technique. Most testing techniques expedite detection of particular error class(es). To achieve extensive error detection capabilities, we must combine techniques that detect different, complementary sets of errors. Moreover, as software must satisfy a variety of qualities, techniques that assess different software qualities must be integrated to meet overall testing objectives.

Multiple testing techniques should be applied in an incremental fashion. It is seldom cost-effective to begin testing by using an expensive, comprehensive technique. Such techniques are better used to demonstrate high reliability after some confidence has been achieved. Otherwise, unnecessary, repeated application of the expensive technique will be required after correcting errors that might have been found with a less costly technique. It is preferred to integrate techniques to provide testing "levels" that range from low cost, low powered capabilities to high cost, high powered capabilities. Higher-cost techniques can profitably use information gained by lower-cost technique application. Effective technique integration enables this approach.

Substantial savings in computation and human effort can be achieved through technique integration that avoids duplication of effort and takes advantage of possible areas of cooperation. Producing the information needed to select test cases is a major cost of any testing technique, regardless of the selection criterion. Integrating techniques requiring some common information allows that information to be shared and reduces cost in technique application. Integrating techniques requiring some common functionality allows tool fragments to be shared and reduces cost in tool development. Testing effort is also reduced by integrating techniques whose criteria might be satisfied by some common test cases. Moreover, explicit technique integration improves software quality by minimizing discontinuities in user activity, which reduces testing efficacy.

We carried out experimentation aimed at evaluating these ideas by integrating DATA FLOW testing and RELAY to achieve adequate code coverage augmented by more comprehensive error detection. These sophisticated testing techniques are prime candidates for integration, because they address different error classes yet share common functionality and use common information. Both are based on the same flow graph representation and both require symbolic evaluation and reasoning capabilities for test data selection. RELAY can, moreover, profitably employ the test cases generated by DATA FLOW testing, which is the less costly technique, thus promoting incremental testing. We hypothesize that software testers will not infrequently encounter the need to integrate such techniques and that effective integration of them can achieve significant runtime efficiencies through the reuse of shared code and shared intermediate results. We therefore performed an experiment to verify this hypothesis.

2 Requirements for Integrating DATA FLOW and RELAY Testing

In our experiment, the testing requirements were that testing provide adequate control and data flow coverage and that comprehensive error detection be provided. We selected DATA FLOW testing to satisfy the first requirement, allowing the user to specify the desired adequacy criterion, and RELAY to satisfy the second, allowing the user to specify fault classes for which error detection must be guaranteed. Thus, our testing requirements are that testing satisfy both the DATA FLOW and the RELAY criteria. We proceeded to develop a testing process that integrates these two techniques. We approached this process development as we would approach the development of product software — namely by identifying requirements, then developing specifications, design, and finally code. In these sections, we summarize our experiences and indicate the software products we produced.

The integrated testing process must meet two sorts of requirements — functional requirements, which are tantamount to the functionality of the two testing techniques themselves, and performance requirements, which require that the integrated process be more efficient (faster and cheaper) than the combination of the individual processes. As will be seen in later sections, this latter performance requirement drives development strongly in the direction of sharing software artifacts that are common to the two techniques. This section describes the functional requirements that characterize DATA FLOW testing and RELAY and indicates performance characteristics that must be achieved in an acceptable integration of the two techniques.

2.1 DATA FLOW Functional Requirements

DATA FLOW testing entails exercising a set of paths that cover particular uses of defined variables. Rapps and Weyuker define a family of criteria for selecting some or all subpaths from a definition to some or all uses of that definition[RW85]. Ntafos' family of criteria requires variable-length chains of alternating definitions and uses [Nta84]. The family of criteria due to Laski and Korel forces the selection of different combinations of definitions that reach a statement, where many variables may be referenced [LK83]. Clarke *et.al.* present a uniform model for defining and comparing the three families[CPRZ86]. Our testing process must develop test path sets that meet one or more of these adequacy criteria and use these paths in the testing process. It should be observed that these DATA FLOW criteria are treated mostly as adequacy measures in past work, but that we consider them as test data selection techniques as well.

A DATA FLOW criterion determines a set of *def-use associations*[1], where a def-use association is a sequence of definitions that must reach uses upon execution of a test datum. In our experiment, we focused on one specific criterion — Rapps' and Weyuker's all-uses. *All-uses* requires coverage of at least one definition-clear subpath from each definition to each use reachable by that definition. Thus, the def-use associations for the all-uses criterion are simply def-use pairs. A *def-use association condition* describes constraints on input data that would execute a definition-clear subpath covering the association. To select test data satisfying a DATA FLOW criterion, all def-use association conditions must be solved.

2.2 RELAY Functional Requirements

RELAY is a model for fault-based testing and analysis [RT88]. Fault-based testing techniques select test data that expedites detection of particular types of faults in source code. Fault-based testing techniques typically generate test data that distinguishes the tested program from alternatives that differ by the defined types of faults. A common assumption is that the tested program is "almost correct" and is faulty by at most a single definable fault[2].

RELAY develops *revealing conditions* that are necessary and sufficient for detection of faults from selected fault classes. A source-cede fault must *originate* an error during execution

[1]The term def-use association is borrowed from Rapps and Weyuker, but can be generalized to refer to Ntafos' chains or the definition combinations of Laski and Korel.

[2]This assumption essentially implies that the faults in the tested program can be detected by distinguishing it from the afore-mentioned alternatives.

that *transfers* through all computations and data flow until it is *revealed* as an observable failure (e.g., upon output or at some other oracle point). For a potential fault, a revealing condition consists of an originating *context error condition* and a *chain transfer condition.* The context error condition guarantees origination (the *origination condition*) and transfer through the computations in the statement containing the fault (the *computational transfer condition*). The chain transfer condition guarantees transfer along some def-use chain (the *data flow transfer condition*) to failure and computational transfer at each use in the chain. The Relay model is described more completely elsewhere [RT88].

A Relay criterion specifies source code locations and class(es) of potential faults that may occur at those locations. For example, one such Relay criterion is variable reference faults for the entire program; another is conditional operator faults in loop conditions. To select test data that guarantees fault detection for a Relay criterion, revealing conditions for each specified potential fault (fault class and location) are generated and then must be solved. In our experiment, we focused on one specific criterion — all variable reference faults.

2.3 DF⋈Relay Performance Requirements

Important hypotheses of this research are that the functional characteristics of Data Flow testing and Relay can be combined to do incremental testing with increased error detection capabilities and that this can be done in such a way that the resulting process is less costly than independently applying the two component techniques. We take this cost characterization as a requirement of our testing process as well.

These performance requirements seemed feasible and generate several subgoals. First, information required by both techniques must be shared rather than generated independently by the techniques. Second, we use the techniques incrementally and require that Data Flow testing be completed before Relay. As a fault-based technique, Relay requires some confidence that the software is "almost correct". By employing Data Flow testing first, we demonstrate this proposition. Third, we require that unnecessary test case generation be avoided by checking those test cases generated by Data Flow testing for reusability by Relay. Since Data Flow testing is computationally cheaper than Relay, this will reduce costs. Fourth, any subfunctions required commonly by the two techniques should be accomplished by shared tool fragments. Fifth, the combination of Data Flow testing and Relay enhances error detection. Data Flow testing expedites detection of erroneous uses of definitions and is complete up to the chosen adequacy criterion. Data Flow does not, however, guarantee that such an error is reflected in the output. Relay guarantees the detection of potential faults identified by the chosen Relay criterion, but this criterion does not necessarily require module "coverage" — that is, only certain statements and fault classes might be selected. Thus, Relay complements Data Flow by more fully testing the actual computations in which a definition is used and ensuring that errors are observed as failures. And, DATA FLOW complements RELAY by providing a more coverage-oriented criterion. Thus, our requirements force the two techniques to be integrated in a way that is more cost-effective than applying both independently.

3 Process Program Specification

In our experiment, we next developed specifications for the integrated testing process. The specification formalism we used to express the functional characteristics of the two major functional components consists of devising a pre- and post-condition pair with a minimal description of functionality

pre-condition: specifies the conditions under which the computation of this functional capability can be used;

post-condition: specifies the state that can be assumed to have been achieved after computation of this functional capability;

function: serves as the high-level specification for the process program to be used to achieve the functional capability.

3.1 DATA FLOW Testing Process Specification

DATA FLOW testing is specified as follows:

pre-condition: DATA FLOW testing can be applied to a module that has been translated to an internal representation and represented by a control flow graph. Any persistent test set (possibly empty) is also input.

post-condition: DATA FLOW testing provides a persistent test set that satisfies a selected DATA FLOW adequacy criterion.

function: For a chosen DATA FLOW adequacy criterion, determine the required def-use associations, mark those covered by data in the persistent test set, and augment the persistent test set to cover the remaining def-use associations.

3.2 RELAY Testing Process Specification

RELAY testing is specified as follows:

pre-condition: RELAY can be applied to a module that has been translated to an internal representation and represented by a control flow graph. Any persistent test set (possibly empty) is also input.

post-condition: RELAY provides a persistent test set that guarantees fault detection for the chosen RELAY criterion.

function: For a chosen RELAY criterion, determine the potential faults, mark those detected by data in the persistent test set, and augment the persistent test set to guarantee detection of the remaining potential faults.

3.3 DF⋈RELAY Process Specification

DF⋈RELAY is specified as follows:

pre-condition: DF⋈RELAY can be applied to a module that has been translated to an internal representation and represented by a control flow graph. Any persistent test set (possibly empty) is also input.

post-condition: DF⋈RELAY provides a persistent test set that satisfies a selected DATA FLOW adequacy criterion and guarantees fault detection for the chosen RELAY criterion.

function: For a chosen DATA FLOW adequacy criterion, determine the required def-use associations, mark those covered by data in the persistent test set, and augment the persistent test set to cover those remaining. For a chosen RELAY criterion, determine the potential faults, mark those detected by data in the persistent test set, and augment the persistent test set to guarantee detection of those remaining.

4 Process Program Design

We continued to develop the DF⋈RELAY process program by first constructing a high-level design that efficiently and synergistically combines DATA FLOW and RELAY. We then iteratively refined the design by further developing the constituent techniques and their integration. The low-level design identifies common tool fragments and shared objects as well as inter-fragment, inter-object, and object-fragment relationships. Because of this, the design shows how we achieved the required functionality while also meeting performance requirements of lower total testing process execution time as well as the requirements of reduced development costs. We describe the functionality in an informal, structured English PPDL (Process Program Design Language) and show relationships in object/control flow diagrams.

4.1 DF⋈RELAY Process High-Level Design

DF⋈RELAY can not be used unless its pre-condition is satisfied. This pre-condition states that the module is translated into an internal represensation and represented by a control flow graph. Otherwise, front-end analysis tool components are activated to generate the internal representation and control flow graph. Our process program begins with the application of DATA FLOW testing. Given a selected DATA FLOW adequacy criterion, each def-use association that must be covered is identified and a Def-Use association condition is generated. These associations are marked as having been covered or not by the user-selected test data, and each specific test datum that covers the association becomes an attribute of the association. Additional test data is selected to cover the unmarked associations, and the appropriate attribution of data to association is made. Now, RELAY comes into play. Given a selected RELAY criterion, each potential fault (origination location and fault class) that must be detected is identified and a revealing condition is generated. Rather than simply select data for the revealing condition, we determine if any of the previously selected test data satisfies the condition. We need not check all data, only the data attributed to the def-use associations on the transfer route. If Offutt's claim that most data that satisfies the origination condition is effective at transferring the error[3][DGK+88], then this should be an extremely effective, synergistic integration of DATA FLOW testing and RELAY.

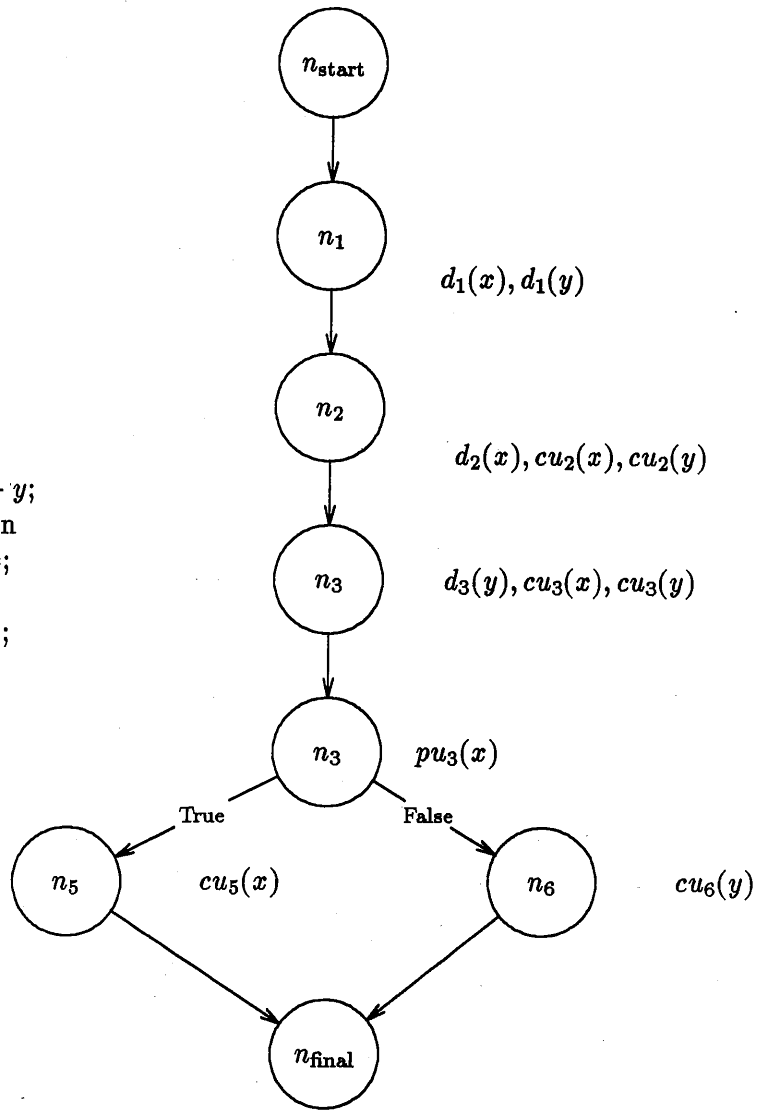
As an example of the integrated use of DATA FLOW testing and RELAY consider the module shown in figure 1. The required def-use associations and conditions for all-uses are shown in Table 1. These def-use associations are covered by the paths $(n_1, n_2, n_3, n_4, n_{final})$ and $(n_1, n_2, n_3, n_5, n_{final})$. A test data set that covers these paths is $\{(x = 4, y = 1); (x = 0, y = 0)\}$. Table 1 shows the attribution of test data to def-use associations. In fact, this set would be a likely choice as most constraint solution schemes select the "simplest" solution[RC85]

Now, suppose that the user selects incorrect variable reference as the fault class. One such potential fault occurs at node n_2, where RELAY postulates that the reference to x should be to y.[4] The origination condition is $(x \neq y)$ and the computational transfer condition at n_2 is $(y \neq 0)$. Thus, the context error condition is $(x \neq y)$ and $(y \neq 0)$. The transfer route for this fault is $(n_2 \ldots n_4)$, and the chain transfer condition is $(x * y = 4)$. The revealing condition, therefore, is $(x \neq y)$ and $(y \neq 0)$ and $(x*y = 4)$. Checking the test data attributed to the def-use associations that intersect this transfer route demonstrates that this revealing condition is satisfied by the test datum (4,1). On the other hand, for an incorrect variable reference to x at node n_3, the transfer route is $(n_3 \ldots n_5)$ and the revealing condition is $(x \neq y)$ and $(x * y \neq 4)$. Checking the test data attributed to the def-use associations that intersect this

[3] Our terminology, not Offutt's

[4] This assumes that x and y are the only variables in the module.

```
n1      input(x,y);
n2      x := x * y;
n3      y := 2 * x + y;
n4      if x = 4 then
n5        output(x);
        else
n6        output(y);
        end if;
```



$d_n(v)$	stands for	**definition** of v at node n
$cu_n(v)$	stands for	**computation-use** of v at node n
$pu_n(v)$	stands for	**predicate-use** of v at node n

Figure 1: Control-flow Graph of Error-Module

def-use association	condition	attribute test data
$d_1(x) \ldots cu_2(x)$	true	(4,1) (0,0)
$d_1(y) \ldots cu_2(y)$	true	(4,1) (0,0)
$d_1(y) \ldots cu_3(y)$	true	(4,1) (0,0)
$d_2(x) \ldots cu_3(x)$	true	(4,1) (0,0)
$d_2(x) \ldots pu_4(x)$	$(x * y = 4)$	(4,1)
$d_2(x) \ldots \neg pu_4(x)$	$(x * y \neq 4)$	(0,0)
$d_2(x) \ldots pu_5(x)$	$(x * y = 4)$	(4,1)
$d_3(y) \ldots cu_6(y)$	$(x * y \neq 4)$	(0,0)

Table 1: Required Def-Use Associations

transfer route demonstrates that this revealing condition has not been satisfied. RELAY must select test data by solving the revealing condition.

Figure 2 provides a high-level design of the DF⋈RELAY process in a structured English PPDL. Figure 3 illustrates this process and the relationships among the tool fragments and objects. Note that this is not strictly a sequential process; many sub-activities can go on in parallel. This parallelism is expressed in the code through control-flow primitive actions which we call triggers. A *trigger* is a signal that causes the activation of a parallel, asynchronous task, which is assumed to run to termination independent of the activity which triggered it.

4.2 DATA FLOW Testing Process Design

We refined our design by developing the major step in the DATA FLOW testing process: the generation of the def-use association conditions. A path selection component selects an initial path (a sequence of nodes through the control flow graph from the start node to the definition) and an activating path for the def-use association (a def-clear sequence of nodes from the definition to the use). A symbolic evaluator interprets each node sequence in terms of symbolic values, providing a path condition and path values that represent the function of the path and is also used to re-evaluate a path representation in terms of modified symbolic values. A reasoning component checks [path] condition feasibility (possibly incrementally during interpretation). When a condition is infeasible, the path selection component must modify at least one selected path (depending on where and why the infeasibility occurred). Figure 4 shows a high-level design of the def-use association condition generator.

```
-- Determine DATA FLOW Coverage Requirements:
choose DATA FLOW adequacy criterion;
determine def-use associations to be covered;
for each def-use association loop
    generate def-use association condition;
    -- Check Tests for Coverage:
    for each test case in the persistent test set loop
        if test case satisfies the def-use association condition then
            mark def-use association as satisfied by test case;
        end if;
    end loop;
    if def-use association not marked as satisfied then
        -- Additional Testing Required:
        select test data that satisfies the def-use association condition;
        mark def-use association as satisfied by this test case;
        trigger testing for newly-selected test case;
    end if;
end loop;
-- Determine RELAY Requirements:
choose RELAY criterion;
determine potential faults to be covered;
for each potential fault loop
    generate revealing condition;
    -- Check Tests for Coverage:
    for each test case marked by the first def-use association
            along the transfer route loop
        if test case satisfies the revealing condition then
            mark revealing condition as satisfied by test case;
        end if;
    end loop;
    if revealing condition not marked as satisfied then
        -- Additional Testing Required:
        select test data that satisfies the revealing condition;
        mark revealing condition as satisfied by this test case;
        trigger testing for newly-selected test case;
    end if;
end loop;
```

Figure 2: High-Level Design of the DF⋈RELAY Process

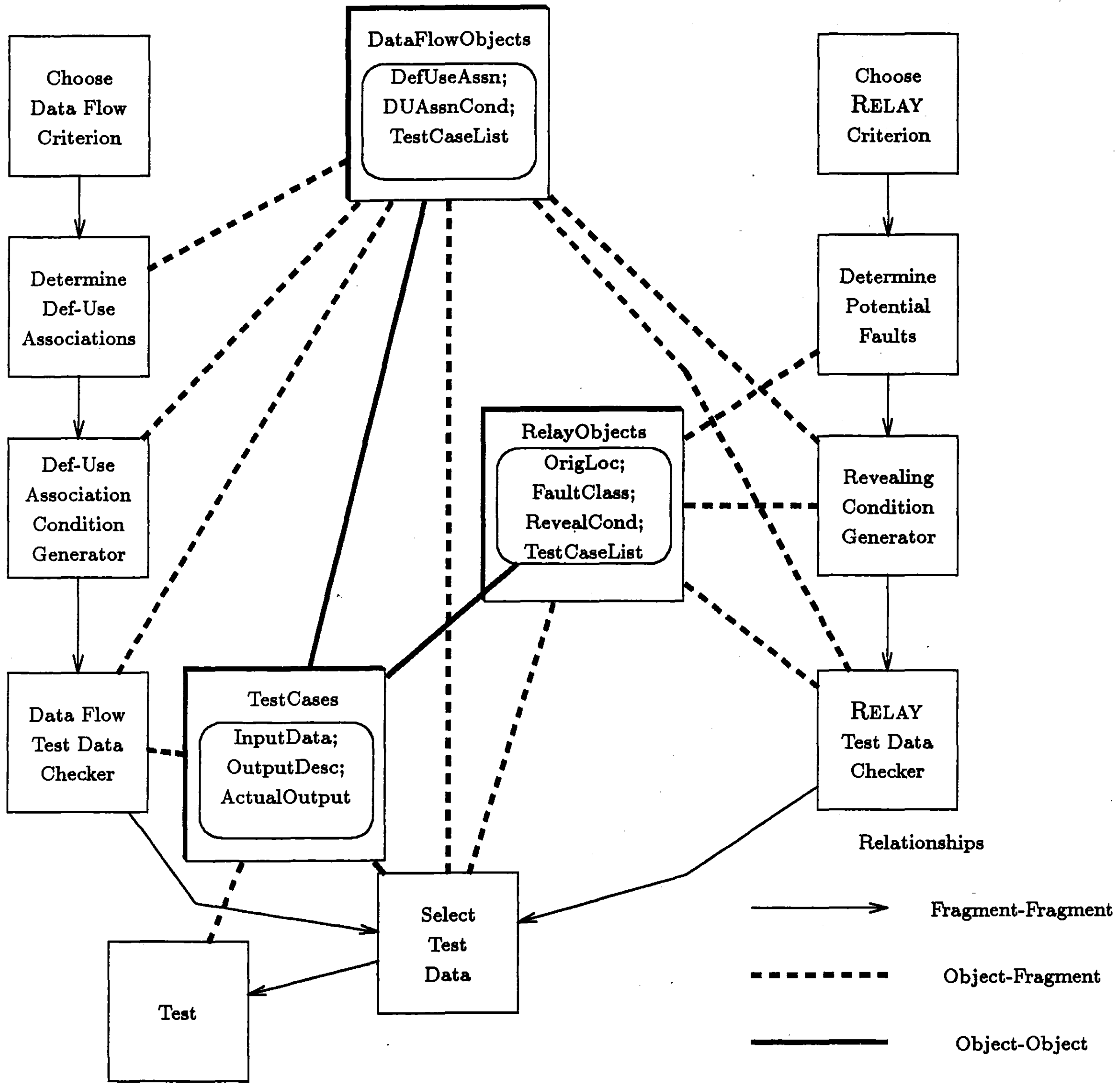


Figure 3: DF⋈RELAY Process

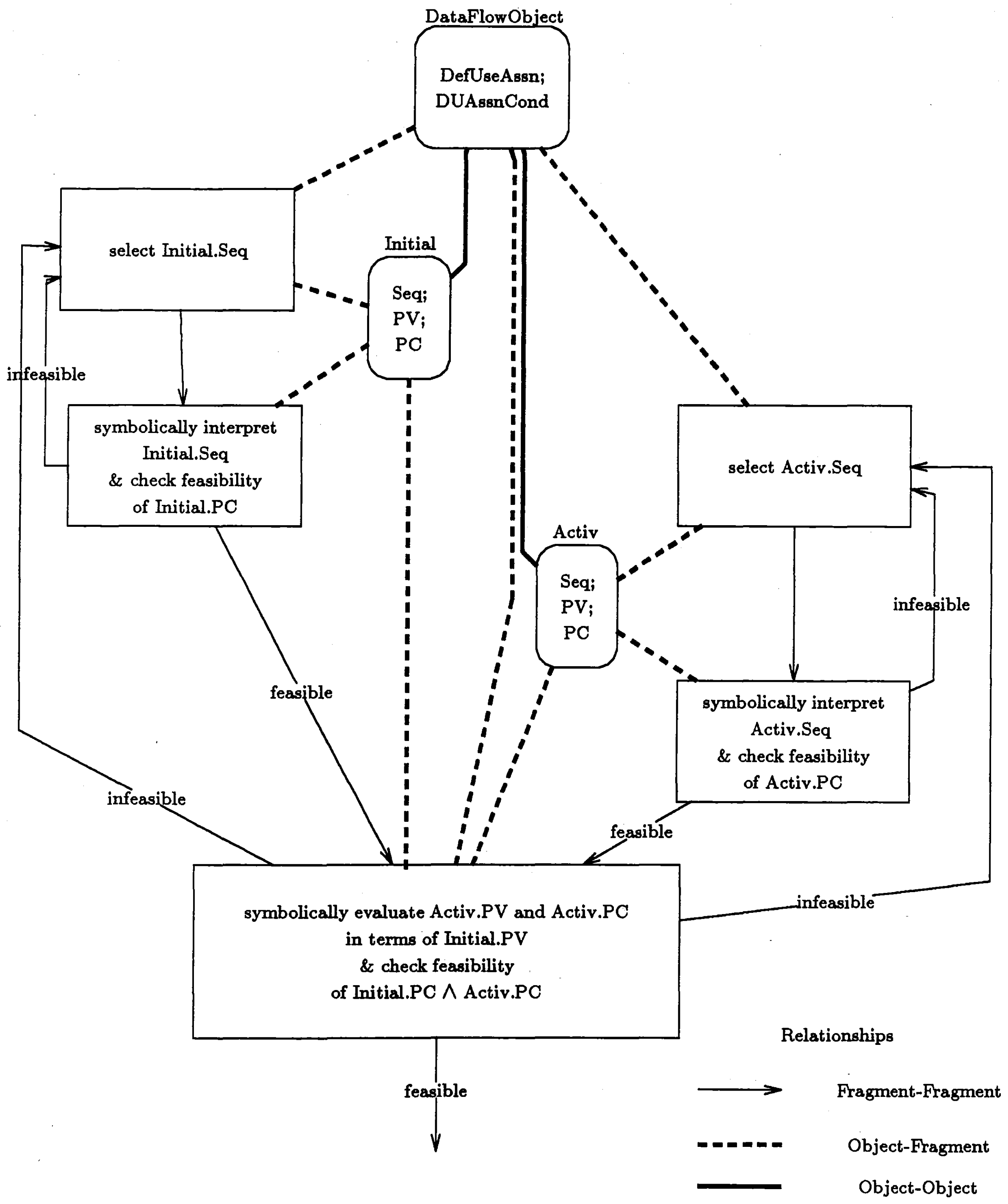


Figure 4: DATA FLOW: Def-Use Association Condition Generator

4.3 RELAY Testing Process Design

We also refined the major step in the RELAY process: the generation of revealing condition generations. As with the similar step in the DATA FLOW process, a path selection component selects an initial path, a transfer route (a sequence of nodes through which the fault transfers to output), and an activating path for the transfer route. A symbolic evaluator interprets these paths and later re-evaluates them. A reasoning component checks condition feasibility. Figure 5 shows a high-level design of the revealing condition generator.

4.4 DF⋈RELAY Process Low-Level Design

In developing the modular decomposition of the DF⋈RELAY process, we identified the following major functional and data modules. Note that many of these are shared by the two techniques.

Functional Modules

- Internal representation translator: must be triggered if the source module has not yet been translated to internal representation
- CFG Generator: must be triggered if the source module has not yet been represented as a control flow graph;
- Path Selector: given nodes to be traversed (e.g., initial node to internal node, def-use pair, transfer route) selects a sub-path covering those nodes;
- Symbolic Evaluator: interprets a path in terms of symbolic values or re-evaluates a symbolic expression in terms of other symbolic values;
- Reasoning Component: checks feasibility of a condition, checks input data for condition satisfaction, and selects input data that satisfies a condition;
- DATA FLOW Testing: allows user to "start up" DATA FLOW testing by choosing a DATA FLOW adequacy criterion, determines the required def-use associations, and generates the def-use association conditions;
- RELAY: allows user to "start up" RELAY by choosing a RELAY criterion, determines the potential faults, and generates the revealing conditions;

Data Modules

- attributed parse tree representation: internal form of the source code (accessed by virtually all components);

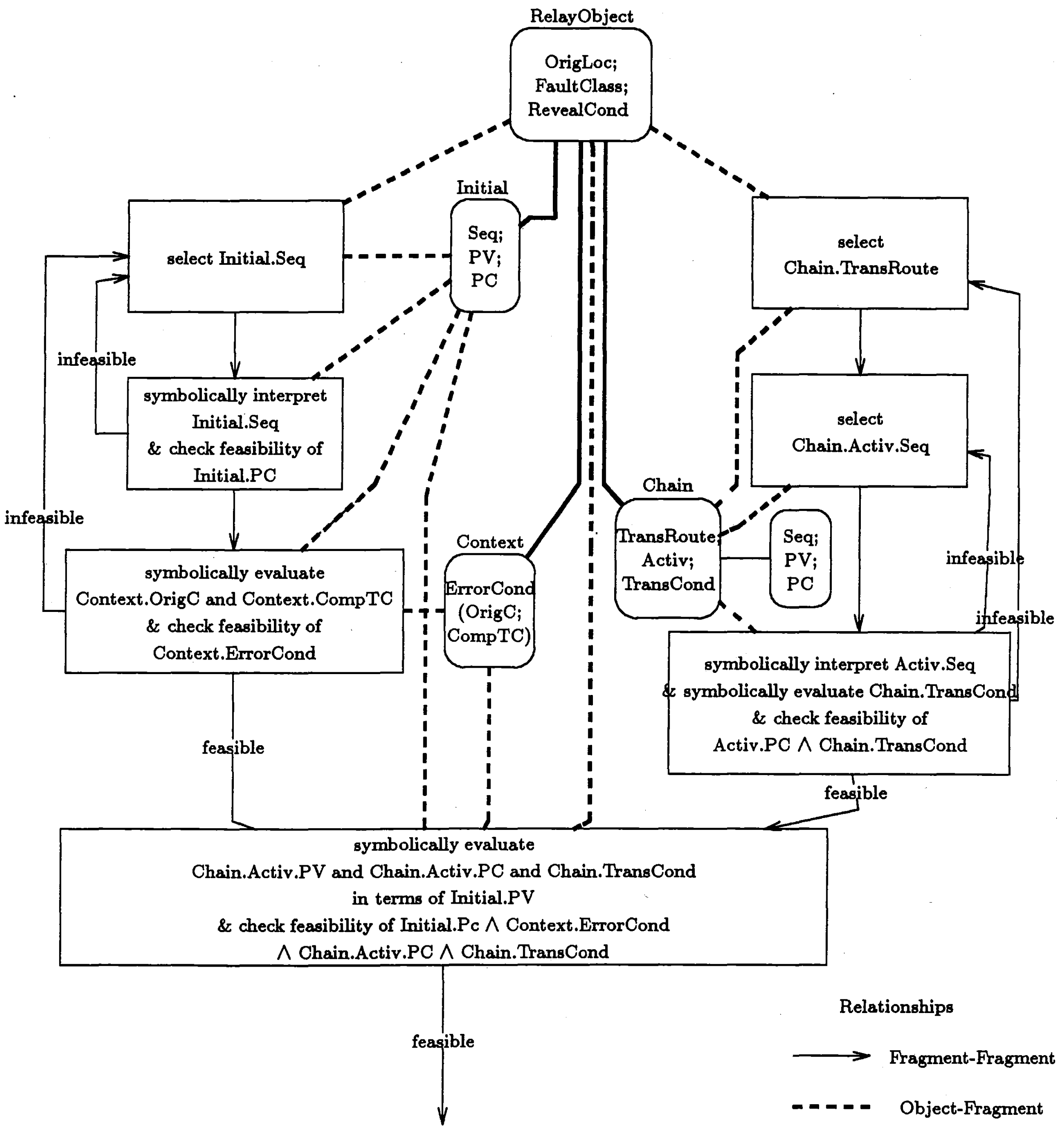

RelayObject
OrigLoc; FaultClass; RevealCond
Initial
Seq; PV; PC
select Initial.Seq
infeasible
symbolically interpret Initial.Seq & check feasibility of Initial.PC
infeasible
symbolically evaluate Context.OrigC and Context.CompTC & check feasibility of Context.ErrorCond
Context
ErrorCond (OrigC; CompTC)
Chain
TransRoute; Activ; TransCond
Seq; PV; PC
select Chain.TransRoute
select Chain.Activ.Seq
infeasible
infeasible
symbolically interpret Activ.Seq & symbolically evaluate Chain.TransCond & check feasibility of Activ.PC ∧ Chain.TransCond
feasible
feasible
symbolically evaluate Chain.Activ.PV and Chain.Activ.PC and Chain.TransCond in terms of Initial.PV & check feasibility of Initial.Pc ∧ Context.ErrorCond ∧ Chain.Activ.PC ∧ Chain.TransCond
feasible
Relationships
Fragment-Fragment
Object-Fragment
Object-Object

Figure 5: RELAY: Revealing Condition Generator

- Control Flow Graph Representation: graph representation of the source code (accessed by virtually all components);
- DATA FLOW Objects: def-use association, def-use association condition, and relationship to test case;
- RELAY Objects: potential fault, revealing condition, and relationship to test case;
- Persistent Test Set: maintains all generated test cases, test case execution results, and relationships to test requirements;
- Test Harness: surrounds the module and executes it on a test case, updating the persistent test set with actual output, and determining if an error has been revealed.

The Appendix provides the code for part of the Data Flow $\bowtie$ RELAY process program.

5 Process Program Analysis

We used process programming to explicitly integrate DATA FLOW testing and RELAY as an experiment. We wanted to study how process programming could be used to guide the development of customized testing processes to meet specific stated testing requirements. The development of the process program helped us devise a hybrid testing process which combines DATA FLOW testing and RELAY in a highly efficient way, that enables effective reuse of intermediate data and common subprocessing steps.

Our process program satisfies the following goals:

1. Information required by both techniques is shared;
2. DATA FLOW testing is completed before RELAY;
3. Tests are not duplicated and unnecessary tests are not generated;
4. Similar subactivities are accomplished by shared tool fragments;
5. Error detection is increased.

We observe a number of benefits arising from the use of software development techniques to create this process program. In that we began by studying the requirements for the synthesized process, we were forced to make our testing goals explicit. Thus we wered precise about just what sorts of efficiencies we were attempting to gain by the synthesis of the two testing techniques. While we believe that significant efficiencies were achieved, further evaluation of DF$\bowtie$RELAY and comparison with the runtime characteristics of the individual techniques are necessary. These further evaluations are tantamount to testing and evaluation of the DF⋈Relay testing process. It is possible for this evaluation to be far more precise and definitive because our process development technique produced a requirements specification against which to evaluate our process. If it turns out that our process program is not optimal with respect to these requirements, we believe that our process will be easy to improve by using software maintenance techniques.

We also note that the development of an actual, tangible process program facilitates empirical evaluation of technique efficacy—both for the individual techniques and for the integrated technique. A problem with previous attempts at empirical evaluation of testing techniques has been that such evaluations have tended to be quite subjective. This is particularly true of combinations of techniques. Our evaluations of such characteristics as speed of testing processes and tools greatly facilitated and rigorized by the fact that we can readily treat this activity as a code instrumentation and monitoring activity.

Another advantage of process programming is that it provides clear incentives to halting the creation of larger, clumsier, and more overloaded testing tools. Software testing tool developers have long recognized that testing needs are large and demanding. Their response has typically been to build growing, monolithic tools and techniques. We believe that the proper solution to the problem of meeting diverse testing needs is the development of process programs, constructed out of carefully designed, well-engineered, test tool modules. This research indicates how this might be done and, we believe, also indicates the subtle, but important, difference between devoting programming effort to constructing monolithic tools and devoting programming effort to developing testing processes.

In continuing the development of larger, more complex tools, test tool developers pursue an essentially bottom-up approach to the problem of meeting diverse testing needs. They make it increasingly difficult to pick and choose from the constituent capabilities of the systems they build, as the systems are becoming more opaque and more tightly bound together internally. Process programming is a top-down approach to meeting software testers needs. It obliges testers to consider testing requirements, design effective solutions, and then code these solutions. Ultimately, this approach will fail if software testers have to design and code these solutions from scratch. On the other hand, we believe that it is possible for software testers to design these solutions out of reusable test tool modules and then assemble their custom testing processes by some modest amount of programming, with heavy reuse of test tool modules. In the described experiment, we found it rather easy to identify the modules needed to meet our needs. We believe that similar experiments can and should point the way to the assembly of a basis set of small, flexible, efficient modules that would support a wide variety of testing objectives when reassembled under the guidance of a process programming approach.

The very fact that software testing process programming compels software developers to think seriously early in the development lifecycle seems important, in itself. Software testing is often slighted in the software development process. This has been attributed to a variety of factors. We believe that an appropriate testing process program approaches solutions to many of these. A poor understanding of what testing should include is often the cause of confusion during the testing phase. A testing process program makes the scope of the testing process explicit. Failure to plan for the testing phase often results in over confidence in the results of a few good test runs. A testing process program forces the user to plan for testing and makes it easy to see when the testing actually performed is less than what was originally planned. The expense and human intensity of the testing process often causes the testing phase to be cut short due to cost overrun. A testing process program can be designed to reduce overall testing costs through effective reuse of testing artifacts and processing steps. In addition, a testing process program can be designed to be highly proactive, thereby requiring less human effort. The process program can supervise much of the iterative, mechanical application of tools, storing of intermediate data and comparison of test results. As these are tedious activities, the use of the process program can potentially reduce the cost and improve the quality of the testing process itself. In addition, a testing process can be programmed to automatically trigger sequences of testing processes without human intervention and could possibly be used to optimize use of computing resources. For example, we are beginning experimentation with a process program designed to trigger regression testing whenever a software change is made.

6 Conclusion

We have presented the results of an experiment to use process programming to integrate DATA FLOW testing and RELAY. The process program we developed makes explicit the synergistic application of the two techniques and management of the test objects produced by both techniques. Our process program was developed using a traditional software development approach entailing the writing of requirements, specifications, design, and code, as well as evaluation and maintenance. Our analysis of this experiment highlights numerous advantages of using process programming to integrate these techniques. Our experience encourages us to believe that this approach is more generally useful, and we intend to use it to develop a variety of other customized testing processes. It is significant to note that the TEAM testing environment, which is currently under development, also plans to use process programming to manage the integration of a diverse assortment of testing techniques [CRZ88].

We see this work as leading to the definition and implementation of a growing library of small, flexible, basic testing tool modules. While the one process program described here is specific to DATA FLOWtesting and RELAY, it is clear that large portions of it may be re-used.

In our design, we successfully isolated generic components and reusable information. In the future, we expect that, as we program increasing numbers and varieties of testing processes, we will identify a growing collection of such testing tool modules. This collection should grow into a library that is widely applicable in facilitating the development of other customized testing process programs.

We intend to explore the development of software environment infrastructure support mechanisms to facilitate the efficient execution of such testing processes. In particular, through the Arcadia software environment research project we are developing object management capabilities which should be effective in supporting the manipulation of software test objects [TBC+88]. Automatic triggering capabilities, which will support the execution of increasingly pro-active testing processes in which changes automatically trigger appropriate activities. For example, we are developing testing process programs which specify that a change in a software product (eg. code or requirements specification) automatically triggers the generation of new or altered test case sets; and that a change in a test case set automatically triggers test runs on the new test cases.

We believe that the results of our first experiment are encouraging, but that more such work is needed. We will be continuing to develop testing process programs. We expect to learn which techniques are most profitably integrated, what process programs should look like, what characteristics process programming languages should have, how well people interact with process programs, and what a comprehensive library of test tool modules should contain. We expect to provide more examples of custom testing process programs and evaluations of their strengths and weaknesses.

References

[CPRZ86] Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. An investigation of data flow path selection criteria. In *Proceedings of the ACM SIGSOFT/IEEE Workshop on Software Testing*, pages 23–32, Banff, Canada, July 1986.

[CRZ88] Lori A. Clarke, Debra J. Richardson, and Steven J. Zeil. TEAM: A support environment for testing, evaluation, and analysis. In *Proceedings of ACM SIGSOFT '88: Third Symposium on Software Development Environments*, pages 153–162, Boston, MA, November 1988.

[DGK$^+$88] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An extended overview of the mothra software testing environment. In *Proceedings of the ACM SIGSOFT/IEEE Second Workshop on Sofware Testing, Analysis and Verification*, Banff, Canada, July 1988. IEEE.

[LK83] Janusz W. Laski and Bogdan Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, SE-9(3):347–354, May 1983.

[Nta84] Simeon C. Ntafos. On required element testing. *IEEE Transactions on Software Engineering*, SE-10(6):795–803, November 1984.

[Ost87] Leon Osterweil. Software processes are software too. *9th International Conference on Software Engineering*, 1987.

[RC85] Debra J. Richardson and Lori A. Clarke. Testing techniques based on symbolic evaluation. In T. Anderson, editor, *Software: Requirements, Specification and Testing*, pages 93–110. Blackwell Scientific Publications Ltd., 1985.

[RT88] Debra J. Richardson and Margaret C. Thompson. The RELAY model of error detection and its application. In *Proceedings of the ACM SIGSOFT/IEEE Second Workshop on Software Testing, Analysis and Verification*, Banff, Canada, July 1988.

[RW85] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

[Sut88] Stanley Sutton, Jr. The APPLA/A programming language background, interim definition, and status. Technical Report CU-88-11, Arcadia, October 1988.

[TBC$^+$88] Richard N. Taylor, Frank C. Bell, Lori A. Clarke, Leon Osterweil, Richard W. Selby, Jack C. Wileden, Alexander L. Wolf, and Michal Young. Foundations for the ARCADIA environment architecture. In *Proceedings of SIGSOFT'88: Third Symposium on Software Development Environments*. ACM, November 1988.

A DF⋈RELAY Process Program Implementation

A.1 APPL/A Overview

Our DF⋈RELAY process program is written in APPL/A [Sut88], which is an extension to Ada. APPL/A provides constructs for the definition of **relations** that represent persistent data and independent, concurrent processes acting on the data. Complete APPL/A programs can be translated into Ada, compiled, and executed.

An APPL/A relation declares a single `tuple` and associated entry and constraint declarations in addition to standard Ada constructs. An `entry` is an operation on a relation; the possible operations are `insert`, `update`, `delete`, `find`, and `select`. A `tuple` element has mode of `in`, `out`, or `in out`. Components of mode `in` are created outside the `tuple`. Components of mode `out` (`in out`) are created (updated) inside of the `tuple` by way of the `determines` clause. Only components of modes `in` and `in out` may be inserted into the `tuple`. A constraint on a relation or on tuples and their attributes are conditions that must be satisfied by the relation.

Section A.2 provides a Uses Chart for the modules specified in section A.3. The APPL/A relations are complete (no bodies are required). We have omitted the entry calls for brevity and because they can be inferred from the tuple declarations. The Ada package specifications are included for the procedural portions of our implementation, but the package bodies have been omitted.

A.2 Uses Chart

The following table shows a uses chart of the functional and data modules described in the low-level design. The code for these modules follows.

Module	Uses
DoDFTieRelay	UserInterface ModuleToBeTested IntRep
ModuleToBeTested	IntRep ControlFlowGraph DataFlow DataFlowObject Relay RelayObject
DataFlowObject	TestCase DataFlow ControlFlowGraph Reasoning PathObject Data
DataFlow	ModuleToBeTested ControlFlowGraph Reasoning DataFlowObject PathObject
RelayObject	ModuleToBeTested Relay ControlFlowGraph TestCase Reasoning PathObject
ContextErrorObject	Reasoning
ChainTransferObject	PathObject Reasoning
Relay	ModuleToBeTested ControlFlowGraph ContextErrorObject ChainTransferObject Reasoning PathObject
Reasoning	Data SequenceOfNodes TestCase
SymEval	SequenceOfNodes Reasoning
PathObject	SequenceOfNodes Reasoning
PathSelector	PathObject
TestCase	Data ControlFlowGraph TestHarness
SequenceOfNodes	ControlFlowGraph
TestHarness	Data
Data	
ControlFlowGraph	IntRep

A.3 APPL/A Code

```
-------------------------------------

 with UserInterface;
 with IntRep;
 with ModuleToBeTested;


 procedure DoDFtieRelay is
   ModuleCode : IntRep.Node;
   DataFlowCriterion : ModuleToBeTested.DataFlowCriterionType;
   RelayCriterion : ModuleToBeTested.RelayCriterionType;


   begin
      -- determines what code to test:
      UserInterface.get(ModuleCode);
      -- determines Data Flow criterion:
      UserInterface.get(DataFlowCriterion);
      -- determines Relay criterion:
      UserInterface.get(RelayCriterion);

      ModuleToBeTested.Insert(DataFlowCriterion,
                              RelayCriterion,
                              ModuleCode);

      -- Data Flow (note that some of this may execute concurrently
      -- with Relay)

          -- Determine Data Flow Adequacy criterion:

          -- determine def-use associations to be covered:
          -- this will be done automatically when a
          -- DataFlowCriterion is inserted.

      -- for each def-use association condition loop:
          -- generate def-use association conditions:
          -- these will be done automatically when the DUAssnGenerator
          -- inserts the association into the tuple.

          -- Check Tests for Coverage:
          -- these will be checked automatically when the
          -- DUAssnGenerator inserts the association into the tuple

          -- Additional Testing Required:
          -- these will be checked automatically when the
          -- DUAssnGenerator inserts the association into the tuple.

          -- Trigger Testing For Each Newly-Selected Test Objects:
```

```
          -- this will be done automatically when any new
          -- test case object is inserted.
      -- end loop

      -- RELAY (note that some of this may execute concurrently
      -- with Data Flow)

          -- Determine Relay criterion:

          -- determine potential faults to be covered:
          -- this will be done automatically when a
          -- RelayCriterion is inserted.

      -- for each potential fault loop:
          -- Generate Revealing Conditions:
          -- this will be done automatically when any new
          -- RelayObjectTuple is created.

          -- Check Tests For Coverage:
          -- these will be checked automatically when the
          -- PotentialFaultGenerator inserts a new tuple.

          -- Additional Testing Required:
          -- this will be done when the
          -- PotentialFaultGenerator inserts a new tuple.

          -- Trigger Testing For  Newly-Selected Test Case:
          -- this will be done automatically when any new
          -- test case object is inserted.

  end DoDataFlowRelay;
---------------------------------------

with IntRep;
with ControlFlowGraph;
with DataFlow;
with DataFlowObject;
with Relay;
with RelayObject;

relation ModuleToBeTested is

  type RelayCriterion is array(natural range <>) of
                              Relay.FaultClassType;

  type DataFlowCriterion is
       (AllDUPaths, AllUses, AllCUses, AllPUses,
        ContextCoverage, OrderedContextCoverage,
        Required2Tuples, Required3Tuples, Required4Tuples);
  type ModuleToBeTestedTuple is tuple
     DFCriterion : in  DataFlowCriterion;
     RCriterion  : in  RelayCriterion;
     ModuleCode  : in  IntRep.Node;
     StartNode   : out ControlFlowGraph.Node;
     FinalNode   : out ControlFlowGraph.Node;
     DFObj       : out DataFlowObject.DataFlowObjectTuple;
     RelayObj    : out RelayObject.RelayObjectTuple;
  end tuple;

determines
   t.ModuleCode determines t.StartNode and t.FinalNode
    by ControlFlowGraph.Create(t.ModuleCode,
                               t.StartNode,
                               t.FinalNode);

  t.DFCriterion determines t.DFObj
    by DataFlow.DUAssnGenerator(t.DFCriterion);

  t.RCriterion determines t.RelayObj
    by Relay.PotentialFaultGenerator(t.RCriterion);

end ModuleToBeTested;
-------------------------------------

with TestCase;
with DataFlow;
with ControlFlowGraph;
with Reasoning;
with PathObject;
with Data;

relation DataFlowObject is

  type DataFlowObjectTuple is tuple
    DefUseAssn   : in  Reasoning.Sequence;
    InitPath     : out PathObject.PathObjectTuple;
    ActivPath    : out PathObject.PathObjectTuple;
    DUAssnCond   : out Reasoning.CNFCondition;
    TestCaseList : in out TestCase.AccessTest;
  end tuple;


  -- A DefUseAssn is a Def-Use Association, which is a path
  -- from a definition to a use.  The DUAssnCond describes
  -- a path from the StartNode to the use.

  -- The DefUSeAssn is inserted by DataFlow.DUAssnGenerator.
-- The DUAssnCond is the combined condition from both the
-- initial path and the active path.

-- Data Flow will search through all of the test case list
-- trying to find a test case that will satisfy the DUAssnCond.

dependencies


  t.DefUseAssn determines t.InitPath
  by PathSelector.SelectSeq(StartNode & t.DefUseAssn.Node,
                            t.InitPath);
  -- determine the Initial Path:
  -- The selector works by selecting a node, calling the
  -- interpreter to interpret that node, and then calling
  -- the feasibility checker to make sure that the subpath
  -- created by the addition of that node is feasible.
  -- Thus, the InitPath returned will be a feasible path.


  t.DefUseAssn determines t.ActivPath
  by PathSelector.SelectSeq(t.DefUseAssn,
                            t.ActivPath);
  -- determine the Activating Sequence:
  -- The selector works the same way for choosing the ActivPath.


  t.InitPath and T.ActivPath determines t.DUAssnCond
    by  DataFlow.DUAssnCondGenerator(t.InitPath,
                                     t.ActivPath,
                                     t.DUAssnCond);
  -- Evaluate ActivPV and ActivPC in terms of InitialPV and
  -- check the feasibility of Initial.PC and Activ.PC
  -- to find the Association Condition.

  t.DUAssnCond determines t.TestCaseList by
  -- iterate over the existing test cases,
  -- if a test case satisfies the DUAssnCond then
  -- add it to the TestCaseList.
    declare
        AccTest : TestCase.AccessTest;
        Data : TestData.Data;
    begin
      for t in TestCase loop
         if Reasoning.Satisfies(DUAssnCond,t) then
            new AccTest;
            AccTest.Test := t;
            AccTest.Next := TestCaseList;
            TestCaseList := AccTest;
         end if;
        end loop;

        if TestCaseList = null then
           new TestCaseList;
           Data := Reasoning.Select(t.DUAssnCond);
           TestCase.Insert(Data,"");
           TestCase.Find(true, Data, "",t.TestCaseList.Test);
           t.TestCaseList.Next := null;
        end if;
    end; -- declare
  end DataFlowObject;
--------------------------------------

with ModuleToBeTested;
with ControlFlowGraph;
with Reasoning;
with DataFlowObject;
with PathObject

package DataFlow is

  procedure DUAssnGenerator(DFCriterion : in
                               ModuleToBeTested.DataFlowCriterion);
  --     Generate the Def-Use Associations.  As each association
  -- is generated it is inserted into a DataFlowObjectTuple.  This
  -- insertion triggers the generation of the Def-Use Conditions.
  -- The exiting test cases are then checked to see if any of them
  -- satisy the conditions. If no test cases satisfy the condition,
  -- then a new test case is generated.


  procedure DUAssnCondGenerator
           (InitialPath : in PathObject.PathObjectTuple,
            ActivatingPath : in PathObject.PathObjectTuple,
            Condition : out Reasoning.CNFCondition);
  --      Determines the condition by
  --  evaluating the Activating Path Value and the
  --  Activating Path Condition in terms of the Initial Path Valu;
  --  and checking the feasibility of the
  --  (Initial Path Condition and Activating Path Condition).

end DataFlow;
--------------------------------------

with ModuleToBeTested;
with Relay;
with TestCase;
with Reasoning;
with PathObject;
with ControlFlowGraph;
```

```
relation RelayObject is


  type RelayObjectTuple is tuple
    OrigLoc      : in  ControlFlowGraph.Node;
    FaultClass   : in  Relay.FaultClassType;
    InitPath     : out PathObject.PathObjectTuple;
    ContEC       : out ContextError.ContextErrorTuple;
    ChainTrans   : out ChainTransfer.ChainTransferTuple;
    RevealCond   : out Reasoning.CNFCondition;
    TestCaseList : out TestCase.AccessTest;
  end tuple;

  -- OrigLoc is the point of the potential fault.

  -- FaultClass is the type of the potential fault.

  -- RevealCond is the revealing condition for that potential fault.

  -- TestCaseList is the test cases that satisfy the revealing condition.
  -- This is of mode out because only those test cases which satisfied
  -- the DUAssnCond for the OrigLoc could possibly
  -- satisfy the Revealing Condition.


  dependencies

    t.OrigLoc determines t.ChainTrans.TransRoute
      by Relay.SelectTransferRoute(t.OrigLoc,
                                   t.ChainTrans);
    -- Note: when an object of type ChainTransferTuple is
    -- inserted, as it is by SelectTransferRoute,
    -- Relay.SelectActivatingSequence is triggered by the
    -- ChainTransfer relation. Therefore, when this trigger
    -- is completed both the TransRoute and the Activating
    -- Sequence will be defined.  After the Activating
    -- Sequence is defined, the ChainTC will be evaluated.
    -- Again, this is being triggered from within
    -- the Chain Transfer relation.

    -- In addition, the SelectSeq procedure may not
    -- be used because the transfer conditions must be
    -- instantiated for each node along the transfer route.


    t.OrigLoc determines t.InitPath
      by PathSelector.SelectSeq(ModuleToBeTested.StartNode
                                    & t.OrigLoc,
                                t.InitPath);
    t.OrigLoc and t.FaultClass and
      t.InitPath determines t.ContEC
      by Relay.GenerateContextErrorCondition(t.OrigLoc,
                                             t.FaultClass,
                                             t.InitPath,
                                             t.ContEC);


    t.InitPath and t.ContEC and t.ChainTrans determines t.RevealCond
      by t.RevealCond := and(t.InitPath.PC,
                                           and(t.ContEC.CompTC,
                                               t.ChainTrans.ChainTC));

    t.RevealCond determines t.TestCaseList by
      declare
          AccTest : TestCase.AccessTest;
          Data    : Data.TestData;
          TestC   : TestCase.AccessTest;

      begin
        for t in DataFlowObject
         where t.DefUseAssn.Node = t.OrigLoc loop
            TestC := t.TestCaseList;
            while  TestC /= null loop
               if Reasoning.Satisfies(RevealCond,TestC.Test.all) then
                  new AccTest;
                  AccTest.Test := TestC.Test;
                  AccTest.Next := TestCaseList;
                  TestCaseList := AccTest;
               end if;
               TestC := TestC.Next;
            end loop;
         end loop;

        if TestCaseList = null then
           new TestCaseList;
           Data := Reasoning.Select(t.RevealCond);
           TestCase.Insert(Cond,"");
           TestCase.Find(true, Cond, "",t.TestCaseList.Test);
           t.TestCaseList.Next := null;
        end if;
      end; -- declare

end RelayObject;
---------------------------------------

with Reasoning;
relation ContextErrorObject is
```

```
  type ContextErrorTuple is tuple
       ContextEC : in Reasoning.CNFCondition;
       OrigC     : in Reasoning.CNFCondition;
       CompTC    : in Reasoning.CNFCondition;
  end tuple;

end ContextErrorObject;
---------------------------------------

  with PathObject;
  with Reasoning;

  relation ChainTrasferObject is
  -- A TransRoute is a series of nodes that may or may not have
  -- edges between them.  The Seq of the ActivPath must be subpath,
  -- i.e. every node in the sequence is connected by an edge to the
  -- next node.

   type ChainTransferTuple is tuple
         TransRoute : in  Sequence;
         ActivPath  : out PathObject.PathObjectTuple;
         ChainTC    : out Reasoning.CNFCondition;
    end tuple;

  dependencies

    -- Select the transfer route
    t.TransRoute determines t.ActivPath
      by PathSelector.SelectSeq(t.TransRoute,
                                t.ActivPath);

    t.ActivPath determines t.ChainTC
      by Relay.GenerateChainTransferCondition(t.ActivPath,
                                              t.ChainTC);
  end ChainTransferObject;
---------------------------------------

with ModuleToBeTested;
with ControlFlowGraph;
with ContextErrorObject;
with ChainTransferObject;
with Reasoning;
with PathObject;

package Relay is


    type FaultClassType is
         (ConstantReferenceFault,
         VariableReferenceFault,
         VariableDefinitionFault,
         BooleanOperatorFault,
         RelationalOperatorFault,
         ArithmeticOperatorFault);
     -- See [Rich86] for a description of these faults.


procedure PotentialFaultGenerator
          (RCriterion : in out ModuleToBeTested.RelayCriterion);
--      This generates all the potential faults and
-- inserts them into a RelayObject.  It is called from
-- ModuleToBeTested.

procedure GenerateContextErrorCondition
         (OrigLoc    : in ControlFlowGraph.Node,
          FaultClass : in FaultClassType,
          InitPath   : in PathObject.PathObjectTuple,
          ContEC     : out ContextErrorObject.ContextErrorTuple);

--      The OrigLoc and the FaultClass are used to determines the
-- the initial path. Once the feasibility of the InitialPC has been
-- checked, the feasibility of the ContextEC (Context Error Condition)
-- is checked. It is called from RelayObject.


procedure SelectTransferRoute
         (OrigLoc : in ControlFlowGraph.Node,
          TransRoute : out ChainTransferObject.ChainTransferTuple);
-- Select a transfer route from the origination location (OrigLoc)
-- to an output.  It is called from RelayObject.

procedure GenerateChainTransferCondition
          (TransRoute : in PathObject.PathObjectTuple,
           ChainTC    : out Reasoning.CNFCondition);

-- Determine the Chain Transfer Conditions from the Transfer Route.

-- In parallel to this, the OrigLoc and FaultClass
-- are also used to determines the Chain (a ChainTransfer tuple).
-- From the Chain, an activating path is selected.

-- When the ChainTC (Chain Transfer Condition, which is
-- the Path Condition and the computational transfer condition)
-- is feasible, and a feasible ContextEC has been found,
-- the ActivPV and ActivPC and the ChainTC are all
-- re-evaluated in terms of the InitialPV (Initial Path Value).

-- If InitialPC and ContextEC and ActivPC and ChainTC are all
```

```
  -- feasible then a revealing condition has been found.  Otherwise,
  -- either pick a new initial path, or a new transfer route, or both.

end Relay;
---------------------------------------

with Data;
with SequenceOfNodes;
with TestCase;

package Reasoning is

  type CNFCondition is private;
  type Value is private;

  function Select(Condition : in CNFCondition) return Data.Datatype;
  -- Returns data that will satisfy Condition.  Unsolvable is
  -- raised if Condition is not Feasible.

  unsolvable : exception;
  -- No data could be found to satisfy the CNFCondition

  function Feasible(Condition : in CNFCondition) return boolean;

  -- Returns FALSE if Condition describes the empty set;
  -- otherwise it will be TRUE.

  function Satisfies(Condition : in CNFCondition,
                     TheData   : in Data.Datatype) return boolean;
  -- Returns is TRUE if TheData satisfies Condition.
  private

     -- CNFCondition is a Conjunctive Normal Form Condition
     -- i.e. (a v b v c)(d v e)

     -- Value is the symbolic value of a path.
     -- Each variable is described by a syntax tree of the operations upon
     -- that variable.

  subtype SymVar is string;
       -- for now we will keep the symbolic variables represented
       -- as strings.  We expect to use a symbol table package
       -- in the future.

  type SymValNode;

  type AccessSymValueNode is access SymValNode;

  type SymValNode is record
    ValueNode : IntRep.Node;
    Next : AccessSymValueNode;
  end record;

  type VarValuePair is record
        Var : SymVar;
        Val : SymValueNode;
   end record;

  type Disjunct is array(natural range <>) of SymValNode;

  type CNFCondition is array (natural range <>) of Disjunct;

  type Value is array (natural range <>) of VarValuePair;
end Reasoning;
---------------------------------------

with SequenceOfNodes;
with Reasoning;
package SymEval is

  procedure Interpret(Path          : in SequenceOfNodes.Sequence,
                      PathValues    : out Reasoning.Value,
                      PathCondition : out Reasoning.CNFCondition);
  -- Symbolically interprets a path providing
  -- the path value and the path condition.

  procedure Eval(Values   : in Reasoning.Value,
                 PathVal  : in out Reasoning.Value,
                 PathCond : in out Reasoning.CNFCondition);
   -- Symbolically evalute PathVal and PathCond in terms of Values.

end SymEval;
---------------------------------------

with SequenceOfNodes;
with Reasoning;
with SymEval;

relation PathObject is

  type PathObjectTuple is tuple
     Seq : in SequenceOfNodes.Sequence;
     PV  : out Reasoning.Value;
  --      The PV is the set of symbolic values for the variables in the
  --   Seq(uence).
     PC  : out Reasoning.CNFCondition;
  --      The PC is the path condition for execution of
  --  the nodes in Seq(uence).
  end tuple;
  dependencies
     t.Seq determines t.PV and t.PC
       by SymEval.Interpret(t.Seq, t.PV, t.PC);
     -- Execute the sequence to determine the path value and
     -- the path condition.

  constraints
     every t in PathObject satisfies
       (t.Seq /= null);
     -- There is always a sequence
     end every;

end  PathObject;
---------------------------------------

with PathObject;
package PathSelector is

   type Sequence;
   type AccessSequence is access Sequence;
   type Sequence is record
       Node : ControlFlowGraph.Node;
       Next : AccessSequence;
   end record;
   -- Note: A sequence does not necessarily define a complete path.

  procedure SelectSeq(Skeleton : in NodeSeq,
                      Path     : out PathObject.PathObjectTuple);
  --     From a skeleton (a sequence of def - use,def - use,def...use)
  -- select a complete path.

end PathSelector;
---------------------------------------

with Data;
with ControlFlowGraph;
with TestHarness;

relation  TestCase is

  type  TestCaseTuple is tuple
    InputData         : in  Data.DataType;
    OutputDesc        : in  Data.DataType;
    ActualOutput      : out Data.DataType;
  end tuple;

  type AccessTest is access TestCaseTuple;
  type ListOfTests;
  type AccessTest is access ListOfTests;
  type ListOfTests is record
```

```
    Test : AccessTestTuple;
    Next : AccessTest;
  end record;


  dependencies
    t.InputData and t.OutputDesc determines t.ActualOutput
      by TestHarness.Execute(t.InputData,
                             t.OutputDesc,
                             t.ActualOutput);

  constraints
    every t1 in TestCase satisfies
        no t2 in TestCase satisfies
           (t1.InputData = t2.InputData) and
           (t1.OutputDesc = t2.OutputDesc);
        end no;
    end every;
end TestCase;
---------------------------------------

with ControlFlowGraph;
relation SequenceOfNodes is
  -- This defines a complete sequence
   type Sequence is tuple
      Node : in ControlFlowGraph.Node;
      Edge : in ControlFlowGraph.Edge := null;
      NextNode : in ControlFlowGraph.Node;
   end tuple;

constraints
   every t in Sequence satisfies
     if t.Edge /= null then
       t.Edge.FromNode = t.Node;
       t.Edge.ToNode = t.NextNode;
     end if;
  end every;

end SequenceOfNodes;
---------------------------------------

with Data;
package TestHarness is

  procedure Execute(InputData    : in Data.Datatype,
                    OutputDesc   : in Data.Datatype,
                    ActualOutput : out Data.Datatype);

    IllegalResult :  exception;
    --      The module, when run on the InputData,
    -- returns a value in conflict with the
    -- OutputDesc.

    NeverHalts    :  exception;
    --      The module, when run on the InputData,
    -- does not halt within the time limit;

    TypeConflict  :  exception;
    --      The types of InputData or OutputDesc do
    -- not match the types of the inputs or outputs of
    -- the module.

  end TestHarness;
---------------------------------------


package   Data is
  -- TypeDefinitions is defined in Debus, page 4.
  -- ImageOfValue is a variable length string which is the
  -- image of the value.
  type  ConstraintClass is (RangeClass, ValueClass);
  type  Datatype (Constraint : ConstraintClass := RangeClass) is
    record
      case  Constraint is
        when  RangeClass =>
               TypeofRange : TypeDefinitions;
               StartRange  : ImageOfValue;
               EndRange    : ImageOfValue;
        when ValueClass =>
               TypeofValue : TypeDefinitions;
               TheValue    : ImageOfValue;
      end case;
    end record;
end Data;
---------------------------------------

with IntRep;
package ControlFlowGraph is
-- This is a very simplified form of the proposed
-- ControlFlowGraph package.

   subtype Node is IntRep.NodeNumber;
   type EdgeRecord;
   type Edge is access EdgeRecord;
   type EdgeRecord is record
      FromNode : Node;
      ToNode   : Node;
    end record;

   procedure Create(ModuleCode : in IntRep.Node,
                    StartNode  : out Node,
                    FinalNode  : out Node);
  -- This creates a Control Flow Graph representation of the module
  -- code.
end ControlFlowGraph;
```