Data Flow Based Integration Testing *

Plínio R. S. Vilela[1]
José C. Maldonado[2]
Mario Jino[3]

[1]Telcordia Technologies, Inc. - formerly Bellcore
445 South Street,
Morristown – NJ 07960, USA
vilela@research.telcordia.com

[2]ICMC-USP
1465 Dr. Carlos Botelho ave.,
São Carlos - SP 13560-970, Brazil
jcmaldon@icmc.sc.usp.br

[3]DCA/FEEC/Unicamp
400 Albert Einstein ave.,
Campinas – SP 13083-970, Brazil
jino@dca.fee.unicamp.br

Abstract

An approach to extend Structural Unit Testing Criteria to the *Integration Testing* is presented. The approach is based on the concept of *Potential Uses*, which basically states that a data flow association exists between a definition of a variable and every point reachable from that definition through a *definition-clear subpath*, even without an explicit usage of the variable – it is called a *potential data flow association*. A practical procedure to implement the approach, based on testing information generated during unit testing, is also presented.

KEY WORDS: Software Testing, Data Flow, Integration Testing.

---

*Partially supported by Fapesp, CNPq, CAPES and Fulbright.
Copyright 1999 Telcordia Technologies, Inc.
All rights reserved.

1 Introduction

Integration testing and *unit testing* are meant to test different aspects of the software. Even if unit testing were applied to the furthest possible extent and the behavior of a module M was guaranteed to be correct for its entire input domain, as far as M is used with other units a communication protocol is established and this communication still needs to be tested.

During development, programs must be examined in order to check whether they do what they are supposed to and do not do what they are not supposed to. For that matter the program must be accompanied by a description or specification stating what is correct and what is incorrect.

When testing, the consistency of the program is examined through a subset of the input domain. It is generally impossible to use the entire input domain to test the program and the subset used is much smaller than the entire set. What is implicitly assumed is that if the program performs well on the selected subset it will perform well on the entire set. Actually this is one of the weak points of testing, since the reliability of the statement on the quality of the program depends on the adequacy of the selected test data set. Thus, a key problem of program testing is the selection of the test data set and its quality.

There are two well known strategies for selecting test data: *black box* (or *functional*) testing and *white box* (or *structural*) testing. On *Functional Testing* test data is derived exclusively from the specification; specially observed are the functions and sub-functions of the program, the domain and sub-domains and the function's range. On *Structural Testing* test data is derived from the analysis of the program structure, paths, subpaths, data flow, and the expressions and data values.

Data Flow based Testing is a structural testing technique which concentrates on the analysis of data flow information to derive testing requirements. The basic idea is to reduce the number of required paths by looking at those subpaths that start on a statement where a value of a variable is set or changed and finally reach a statement where this value is used or is available to be used [1].

Many data flow based selection criteria are known today. However, most of them are meant to be applied for unit testing. Extending these criteria to integration testing is the main concern of many researchers; and, as well, the major concern of this work.

According to Pressman [9], *Integration Testing is a systematic technique for constructing tests to uncover faults associated with interfacing. The objective is to take unit-tested modules and build a program structure that has been dictated by design.* A question frequently raised is why further testing is necessary, if all the individual modules have been validated. Still according to Pressman, the problem is "putting them together" – interfac-

[1] The variable definition is alive at the end of the subpath.

ing. *Data can be lost across an interface; one unit can have an inadvertent, adverse affect on another; sub-functions, when combined, may not produce the desired major function; individually acceptable imprecision may be magnified to unacceptable levels; global data structures can present problems.*

Several "evolutions" of unit testing to integration testing have been proposed. *Functional Testing* is directly applicable since the functional specification of the software is available; *Structural Testing* requires some adaptations. In this work two approaches to integration testing are discussed and compared.

The basic idea behind any data flow based approach for integration testing is to use data flow information to help define interprocedural subpaths that must be executed in order to satisfy certain criteria. They basically differ on the way such information is gathered and used, or on the way programs and data flow information are modeled.

This paper presents a family of data flow based criteria to the integration testing of programs called the *Integration Potential Uses Criteria*; this family is based on the concept of potential uses, the criteria is described on Section 3.2. Section 2 presents a terminology used to define the criteria.

Some notions on the implementation of the criteria are presented on Section 4.

2 Terminology

A *module*, in this context, is either a main program or a single subprogram and has only one *entry* and one *exit point*. A module is represented by a directed graph that describes the possible flow of control through the module.

A *control flow graph* or *program graph* of a module M is a directed graph G$(M)=(N, E, n_{in}, n_{out})$, where N is the set of nodes, $E \subseteq N \times N$ is the set of edges, $n_{in} \in N$ is called the *entry node*, and $n_{out} \in N$ is called the *exit node*. Each node in the program graph is associated to a block of statements within the module that are always executed together; that means, when one statement of the block is executed every other statement within the same block is also executed, in the given order. The edges on the program graph represent possible control flow between two diferent blocks. On the definition of the Integration Potential Uses Criteria it is assumed that each procedure call forms its own block.

A program graph defines the paths within a module. A *subpath* in G(M) is a finite, possibly empty, sequence of nodes $p = (n_1, n_2, \cdots, n_{|p|})$ [2] such that for all i, $1 \leq i < \mid p \mid$, $(n_i, n_{i+1}) \in E$. A subpath formed by the concatenation of two subpaths p_1 and p_2 is denoted by $p_1 \cdot p_2$. An *initial subpath* is a subpath whose first node is the entry node n_{in}. A *final subpath* is a subpath whose last node is the exit node n_{out}. A *path* is an initial

[2] $\mid s \mid$ denotes the number of elements in the sequence s.

subpath and a final subpath. The set of all paths in G(M) is denoted by $PATHS(M)$.

A *loop* of a program graph G(M) is a subgraph of G(M) corresponding to a looping construct in module M. All loops have single entry and single exit nodes and they are not part of the loop. Distinctions are made among the several types of subpaths that visit loops. A *cycle* is a subpath of length ≥ 2 that begins and ends with the same node. A cycle $(n) \cdot p \cdot (n)$ such that the nodes of p are distinct and do not include n is called a *simple cycle*. A subpath that is a simple cycle or cycle free is called a *simple subpath*.

The occurrences of variables in a program can either be a definition or a use. Let x be a variable in a module M, a *definition* of x is associated with each node n in G(M) that contains a statement fragment that can assign a value to x; this definition is denoted by $d_n(M, x)$. The set of variables for which there is a definition associated with a particular node n in G(M) is denoted by $\mathit{DEFINED}(M, n)$. A *use* of x is associated with each node n or edge (n, m) in G(M) that contains a statement fragment that can access the value of x.

A variable use is a *computation use*, denoted by c-use$_n(M, x)$, iff the use of variable x in node n directly affects the computation being performed or allows someone to see the result of a previous definition. A variable use is a *predicate use*, denoted by p-use$_{(n_1,n_2)}(M, x)$, iff the use of variable x in (n_1, n_2) directly affects the flow of control within the program. Notice that the c-use is associated with a node and the p-use is associated with an edge. A *definition-clear subpath* with respect to (wrt) a variable x is a subpath p such that for all nodes n in p, $x \notin \mathit{DEFINED}(M, n)$.

A node i has a *global definition* of a variable x if there is a definition of x in i and there is a definition-clear subpath from some node or some edge with a c-use or a p-use of x. A c-use of x in j is a *global* c-use if there is no definition of x in the same node preceding the c-use.

A subpath $(n_1) \cdot q \cdot (n_j, n_k)$ in module M is a *DU-Path* wrt a variable x iff there is a global $d_{n_1}(M, x)$ and: (1) there is a global c-use$_{n_k}(M, x)$ and $(n_1) \cdot q \cdot (n_j, n_k)$ is a simple subpath with $q \cdot n_j$ definition-clear wrt x; or (2) there is a p-use$_{(n_j,n_k)}(M, x)$ and $(n_1) \cdot q \cdot (n_j)$ is cycle free with $q \cdot n_j$ definition-clear wrt x. The same definition of *DU-Path* applies if the subpath q involves a procedure call and the c-use and p-use are of the corresponding formal parameter of x.

A program P is represented by a *direct multigraph* $CG(P) = (\mathcal{N}, \mathcal{E}, s)$, where modules are associated with nodes $n \in \mathcal{N}$ and the possible control flow between modules are associated with edges $e \in \mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$. Since CG is a multigraph, there may be more than one edge between two nodes, if there is at least one edge between M_1 and M_2 there is a *connection* between M_1 and M_2. Every node in the graph can be reached from node s called the *root node*. CG representing a program $\mathcal{P}$ is called a *callgraph*. Each edge $e \in \mathcal{E}$ represents a pair (M_{1_i}, M_{2_i}), $0 < i \leq k$, one of the k calls to module M_2 in module

M_1. Module M_1 is the *calling module* and M_2 is the *called module*. The set of all paths between two modules M_1 and M_2 of CG(P) is denoted by *iPATHS*(M_1, M_2). The set of all paths in the callgraph CG(P) is denoted by *CG_PATHS*(P).

A *path selection criterion for unit testing*, or simply an *unit criterion*, is a predicate that assigns a truth value to any pair (M_1, Π), where M_1 is a module and Π is a subset of *PATHS*(M_1). A pair (M_1, Π) *satisfy* an unit criterion C iff $C(M_1, \Pi)$=*true*. A *path selection criterion for integration testing*, or simply an *integration criterion*, is a predicate that assings a truth value to any pair (P, Ψ), where P is the program and Ψ is a subset of *CG_PATHS(P)*; if the integration criterion is pairwise, a *pairwise integration criterion* it is a predicate that assigns a truth value to any triple (M_1, M_2, Φ), where M_1 and M_2 are modules and (M_1, M_2) is a multi edge in CG(P), the *callgraph* of program P, and Φ is a subset of *iPATHS*(M_1, M_2). A triple (M_1, M_2, Φ) satisfy a pairwise integration criterion C iff $C(M_1, M_2, \Phi)$=*true*.

The corresponding formal parameter of an actual parameter x through a call c is denoted as $formal_c(x)$. To simplify the definitions let consider that for a global variable x, $formal_c(x)$ maps the variable to itself; $formal_c(x) = x$.

The node in module M_1 where the call c occurs is denoted by n_c. The set of variables used as input or output on a call c of module M_1 to module M_2 is denoted as: in_c = the set of actual parameters or global variables used as input in the call c of module M_1 to module M_2; out_c = the set of actual parameters or global variables used as output in the call c in module M_1 to module M_2. The set of all $in_c \cup out_c$ is generally called *communication variables*.

A path selection criterion C_1 *subsumes* a criterion C_2 iff every pair (M_1, Π), (P, Ψ) or triple (M_1, M_2, Φ) that satisfies C_1 also satisfies C_2. Two criteria are *equivalent*, in terms of subsumption relation, iff each subsumes the other. A criterion C_1 *stricly subsumes* a criterion C_2 iff C_1 subsumes C_2 but C_2 does not subsumes C_1. Two criteria are *incomparable*, in terms of subsumption relation, iff neither criterion subsumes the other.

3 Integration Testing Criteria

3.1 The Linnenkugel and Müllerburg Criteria

Linnenkugel and Müllerburg [5] worked on the adaptation of a set of well-known testing criteria for unit testing to define similar criteria, based on control and data flow analysis, for integration testing.

Their integration model also represents a program by a *callgraph*. The approach concentrates on analyzing *relations* and *interfaces* between modules. The relations are determined by the call statements in the modules and the interfaces by the data used by both, the calling and the called modules. For testing interfaces they adapted the *Data*

Flow Testing Criteria defined by Rapps and Weyuker [10].

They define a set of criteria based on callgraphs in a similar way as the control flow criteria are defined based on the control flow graph.

Testing Relations, Based on Callgraphs:

- *All-Modules* requires that every module is executed at least once.

This criterion is the analogous of *All-Nodes* defined for program graphs and, as its unit testing counterpart, is possibly the simplest Integration Testing Criteria. It requires that every module in the program is executed at least once, regardless of the complexity or possible combination of module calls within the program.

The good thing is that it is easy to implement, understand and use, but it adds little value on establishing the goodness of a test set. The following example illustrates these aspects. Suppose a program is composed of four modules, a *main* module, a *sort* module and two modules that call the *sort* module. The *sort* module receives an array of numbers, a minimal – *min* and a maximal – *max* number, and returns a sorted array containing elements that are greater than *min*, but less than *max*.

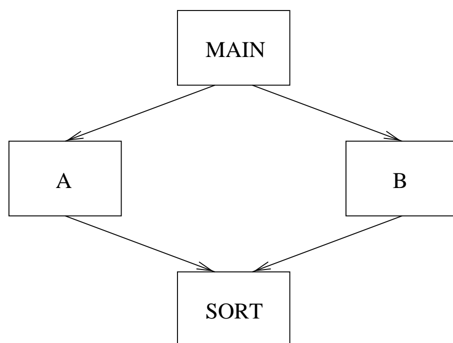

Figure 1: Problems with All-Modules

Considering Figure 1, suppose module *A* calls *sort* correctly – `sort(array,min,max)`, but in module *B* there is a fault, the call inverts *max* and *min* and the statement looks like this `sort(array,max,min)`. Notice that this fault is not easily detected during unit testing, since both module *B* and *sort* could have been implemented by different teams each of which having access only to the specification of the other module [3] (that in this case was misinterpreted).

A test case (or a set of test cases) that executes the sequence (*main*, *A*, *sort*) and (*main*, *B*) would satisfy the *All-Modules* criteria but would not force the execution of (*B*, *sort*) not allowing the fault to be detected.

[3]Beizer [1] classifies this type of fault as `6126` - *Parameter Sequence Fault.*

- *All-Relations* require that the calling relation between modules is executed at least once.

Since the callgraph is a multi-graph there may be more than one edge between two modules, corresponding to more than one call from one module to another; this criterion requires the execution of at least one of such edges. This criterion minimizes the problems with the first one, since it requires every "relation" between modules to be executed, that means at least one call between every two modules must be executed at least once. Although minimized, the problem still exists; suppose there are several calls between B and *sort* and just one is defective, if this is not the call executed by the set of test cases that satisfies *All-Relations* then the fault would still not be detected.

- *All-Multiple-Relation* requires that every call between modules is executed at least once.

This is the analogous of *All-Edges* defined for program graphs, it requires the execution of every call from one module to another within the program, it is the last criterion of this category. This criterion would at least give a chance for the fault presented on the example to be exposed. But still it is not guaranteed, since the test case chosen to execute the defective call could have an empty array as the correct result, in this case the fault would still not be detected.

The following set of criteria were also defined by Linnenkugel and Müllerburg, they go a little further on the concept of traversing a path through the whole program.

- *All-Simple-Call-Sequences* requires that every sequence of (descending) calls without repetition of calls is executed at least once.
- *All-Loop-Iteration-Call-Free-Sequences* requires that every sequence of (descending) calls without repetition of loops is executed at least once.
- *All-Call-Sequences* requires that every sequence of (descending) calls is executed at least once.

Since those criteria do not consider the data flow through the program they are not sufficient to test the interface between modules. The natural evolution, imitating what happened to the unit testing criteria, is to use data flow information and define a new set of Integration Testing Criteria. For that matter the authors defined the following set of criteria, considering the interface between each pair of modules (M_1, M_2) within the program:

Testing Interfaces, Based on Data Flow:

INT-All-Defs requires for every call the execution of a definition-clear subpath wrt each communication variable from each relevant definition to some reference.

- for each $x \in in_c$ a def-clear subpath from each $d_m(M_1, x)$ to some node n with a c-use$_n(M_2, formal_c(x))$, or to some edge (n, n') with a p-use$_{(n,n')}(M_2, formal_c(x))$, has to be executed, and
- for each $x \in out_c$ a def-clear subpath from each $d_n(M_2, formal_c(x))$ to some node m with a c-use$_m(M_1, x)$, or to some edge (m, m') with a p-use$_{(m,m')}(M_1, x)$, has to be executed.

The *INT-All-Defs* criterion is the basic data flow based integration criterion defined. The idea is that if a variable receives a value, then at least one use of this variable must be exercised. This criterion does not provide much improvement over the control flow based criteria previously presented. The next step is to consider each pair definition - use individualy.

INT-All-Uses requires for every call the execution of a definition-clear subpath wrt each communication variable from each relevant definition to every c-use and every p-use.

- for each $x \in in_c$ a def-clear subpath from each $d_m(M_1, x)$ to each node n with a c-use$_n(M_2, formal_c(x))$, and to each edge (n, n') with a p-use$_{(n,n')}(M_2, formal_c(x))$, has to be executed, and
- for each $x \in out_c$ a def-clear subpath from each $d_n(M_2, formal_c(x))$ to each node m with a c-use$_m(M_1, x)$, and to each edge (m, m') with a p-use$_{(m,m')}(M_1, x)$, has to be executed.

The *INT-All-Uses* criterion requires that an interprocedural subpath from each communication variable definition to each c-use and each p-use be executed. It is an interesting criteria, but when the number of possible subpaths between a definition and a use is considered it is easy to see that a lot of testing possibilities are been neglected. Since the number of possible subpaths that can cover a given association is potentially infinite, a DU-Path if considered on the definition of the next criterion as a way to improve the previous criterion but keeping the number of required elements finite.

INT-All-DU-Paths requires for every call the execution of every simple cycle or cycle free definition-clear subpath wrt each communication variable from each relevant definition to every c-use and every p-use.

- for each $x \in in_c$ every du-path from each $d_m(M_1, x)$ to each node n with a c-use$_n(M_2, formal_c(x))$, and to each edge (n, n') with a p-use$_{(n,n')}(M_2, formal_c(x))$, has to be executed, and

- for each $x \in out_c$ every du-path from each $d_n(M_2, formal_c(x))$ to each node m with a c-use$_m(M_1, x)$, and to each edge (m, m') with a p-use$_{(m,m')}(M_1, x)$, has to be executed.

3.2 The Integration Potential Uses Criteria

A variety of Data Flow Based Testing Criteria have been defined in the past few years [3, 4, 8, 10, 11, 6]; previous approaches on Integration Testing are based on the Family of Criteria proposed by Rapps and Weyuker [10]: they define that a data flow association exists between a definition and a further use of a variable. Maldonado, et al. [7, 6] modified this concept and defined a *potential data flow association* introducing a family of data flow criteria called the Potential Uses Criteria. The set of criteria defined in this section is an extension of the Potential Uses Criteria to the integration testing, they are called the *Integration Potential Uses Criteria*.

The approach used to define and apply the criteria is called *pairwise* because it considers the modules in pairs to derive integration testing requirements. The "pairwise" approach supports any incremental integration strategy, e.g. bottom-up, top-down and sandwich, provided that each increment is performed with a pair of modules. An extension to more than two units integration is feasible, but it is not the concern of this work.

INT-All-Potential-Uses:

- for each $x \in in_c$ a def-clear subpath from each $d_m(M_1, x)$ to each node n and to each edge (n, n') with a potential use of $formal_c(x))$, has to be executed, and
- for each $x \in out_c$ a def-clear subpath from each $d_n(M_2, formal_c(x))$ to each node m and to each edge (m, m') with a potential use of x, has to be executed.

The first Integration Potential Uses criterion defined is similar to the previously defined *INT-All-Uses* the basic diference is that now a actual usage of the variable is not required on establishing a interprocedural data flow association, it is only necessary that the variable is alive at a reachable point to make the association be required.

INT-All-Potential-Uses/DU:

- for each $x \in in_c$ a potential DU-Path from each $d_m(M_1, x)$ to each node n and to each edge (n, n') with a potential use of $formal_c(x))$, has to be executed, and

- for each $x \in out_c$ a potential DU-Path from each $d_n(M_2, formal_c(x))$ to each node m and to each edge (m, m') with a potential use of x, has to be executed.

This criterion changes a little bit the way the DU-Path concept is been used. Instead of requiring every such subpaths it requires just one to cover an association.

INT-All-Potential-DU-Paths:

- for each $x \in in_c$ every potential DU-Path from each $d_m(M_1, x)$ to each node n and to each edge (n, n') with a potential use of $formal_c(x))$, has to be executed, and
- for each $x \in out_c$ every potential DU-Path from each $d_n(M_2, formal_c(x))$ to each node m and to each edge (m, m') with a potential use of x, has to be executed.

This criterion is similar to the *INT-All-DU-Paths* but again does not look for the explicity occurrence of a variable usage to establish the association.

3.3 Example

The following example is used to illustrate some of the differences between the family of criteria presented on the previous section.

Considering the program on Figure 2 and the *INT-All-Uses* criterion presented on the previous section, the list of associations required, supposing the call from `compress` to `putrep` on node 5 and variable n, would be from the definitions $d_1(compress, n)$, $d_9(compress, n)$ and $d_{11}(compress, n)$, through the call c_5 to the set of uses:

- p-use$_{(2,3)}(putrep, formal_{c_5}(n))$,
- p-use$_{(2,7)}(putrep, formal_{c_5}(n))$,
- p-use$_{(3,4)}(putrep, formal_{c_5}(n))$,
- p-use$_{(3,5)}(putrep, formal_{c_5}(n))$,
- p-use$_{(8,9)}(putrep, formal_{c_5}(n))$,
- p-use$_{(8,10)}(putrep, formal_{c_5}(n))$,
- c-use$_4(putrep, formal_{c_5}(n))$, and
- c-use$_6(putrep, formal_{c_5}(n))$.

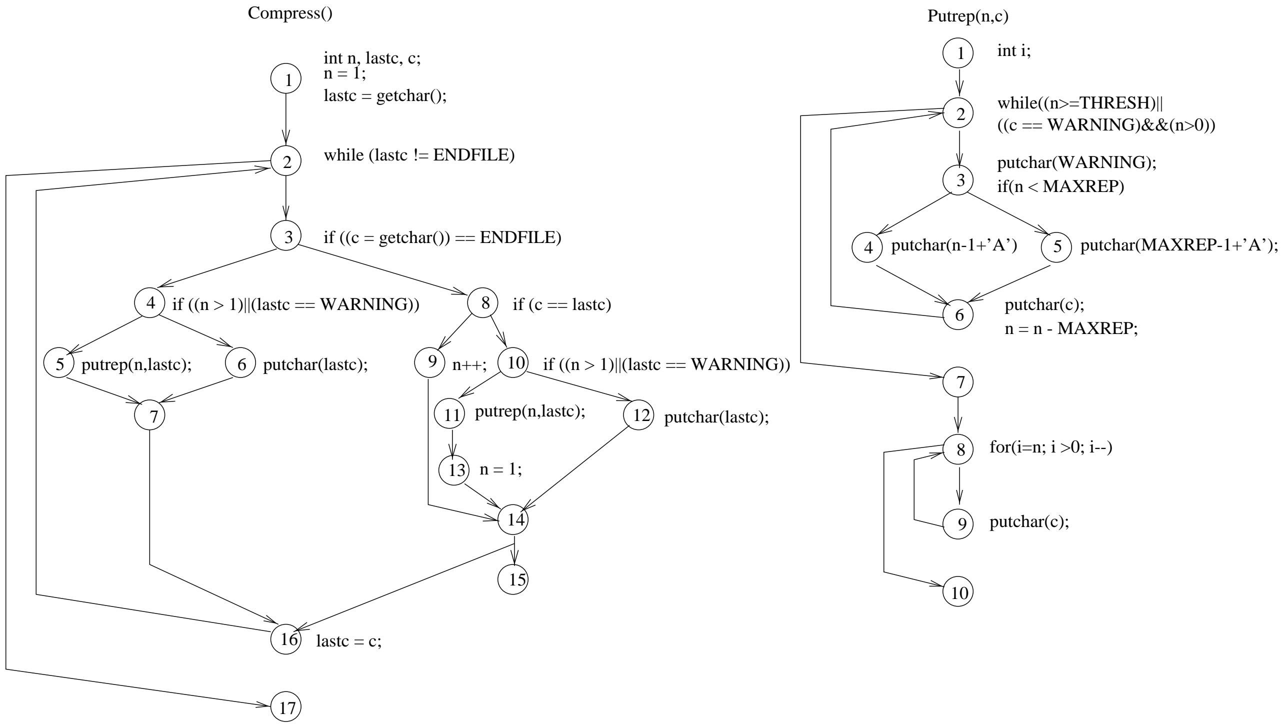

Figure 2: Program Example

This is a very simple example, but even then the *INT-All-Potential-Uses* criterion would require, besides all the associations required by *INT-All-Uses*, associations from the listed three definitions to the potential uses of variable $formal_{c_5}(n)$ on nodes 5 and 9. One could argue that these two nodes would be executed by a test set that satisfies *INT-All-Uses*, well that's true, but the explicity association between one of the variable definitions in `compress` to the potential use on node 5 in `putrep` is not required by *INT-All-Use*, therefore if $formal_{c_5}(n)$ was not used on the boolean expression on the if statement on node 3, the execution of node 5 would not be required.

3.4 Property Analysis

In this section some properties of the criteria presented in the previous section are investigated.

3.4.1 Subsumption Relation

The *subsumption relation* [10] has been used to establish a software testing criteria hierarchy based on the criteria strength. It reflects how easily a test set satisfies a criterion C_2 considering that it has satisfied a criterion C_1.

The subsumption hierarchy presented on [10] defines the relationship for the unit testing criteria. In this section the same relationship is analyzed for the integration

testing criteria. It is assumed that the relations that were not valid for the unit testing criteria remain not valid for the integration testing criteria. Informally it is so because the set of required elements defined by an integration testing criteria is a subset of the required elements defined by the equivalent unit testing criteria if the call sites in the calling module were substituted by the called module code. Consequently it is possible to produce every counter-example needed to prove the not subsumes relations.

To prove the hierarchy presented on Figure 3 it is assumed that every call statement in the program involves at least one communication variable, which are both parameters or global variables. An important difference between the hierarchy presented on [10] for the unit testing criteria and the one presented on Figure 3 for the integration testing criteria is that the Linnenkugel and Müllerburg's data flow integration testing criteria do not subsumes the *All-Multiple-Relations* criterion, while the unit testing equivalents subsume *All-Edges* the equivalent for *All-Multiple-Relations* on the unit testing. Therefore these criteria do not guarantee the execution of every call statement within the program. The following two theorems are presented to demonstrate this characteristic.

Theorem 3.1 *The* INT-All-Uses *criterion and the* All-Multiple-Relations *are incomparable.*

Proof: Let's show that *INT-All-Uses* does not subsumes *All-Multiple-Relations*. It is assumed that in_c is not empty, so let's say $in_c = \{x\}$ where c is the call statement from a module M_1 to a module M_2. Suppose that there are no uses of $formal_c(x)$ in M_2 and no uses of x in M_1 reachable from c, in this case there will be no interprocedural associations between M_1 and M_2 and *INT-All-Uses* will not require c to be executed, which shows that *INT-All-Uses* does not subsumes *All-Multiple-Relations*.

Notice that one can argue that the absence of a variable usage may be a fault; this is an important aspect since programs under test are not considered to be fault free. The situation described above may very well occur in real programs.

Theorem 3.2 *The* INT-All-Potential-Uses *criterion subsumes* All-Multiple-Relations.

Proof: It follows from the proof of Theorem 3.1 that criteria that require the explicitly occurrence of a variable use to establish a interprocedural data flow association, such as *INT-All-Uses* and *INT-All-Defs*, do not subsume *All-Multiple-Relations* criterion. The *INT-All-Potential-Uses* criterion does not require the explicitly occurrence of a variable usage to establish an interprocedural association; consequently, considering in_c not empty for every call c within the program, there will ever be a interprocedural association through c and *INT-All-Potential-Uses* criterion subsumes *All-Multiple-Relations*.

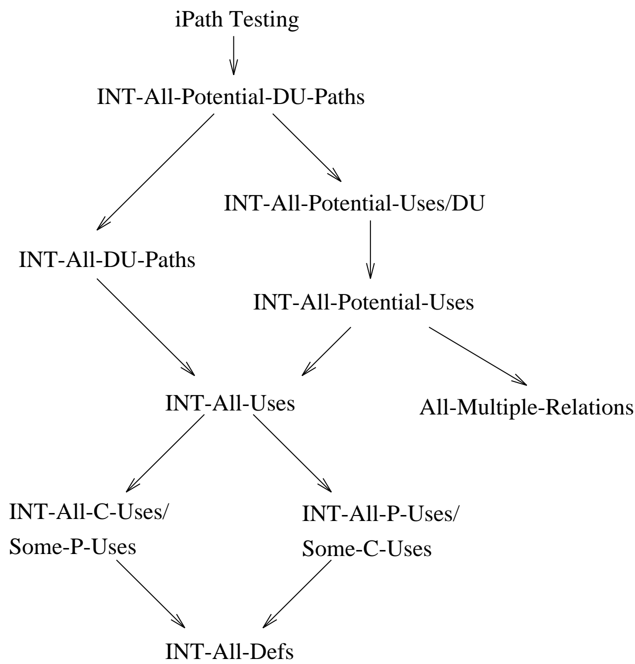

Figure 3: The Subsumption Relation for Integration.

4 Implementation Notions

4.1 Unit Testing

POKE-TOOL [2] is a tool that implements the potential uses criteria for the unit level testing of programs. It has a multi-language architecture based on the usage of an intermediate language and it is currently configured for C, Pascal, COBOL, FORTRAN and Clipper.

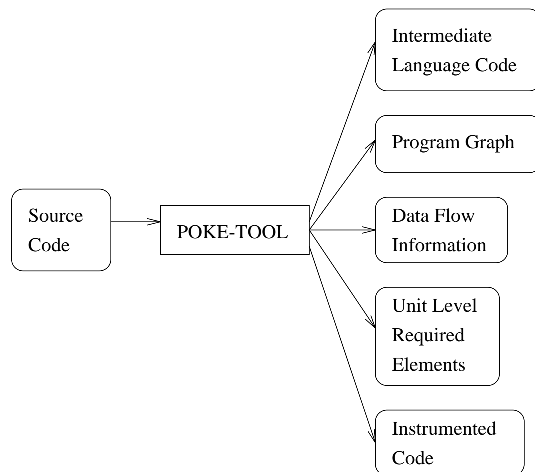

Figure 4: Information Generated by POKE-TOOL.

The tool accepts the source code as input and generates the intermediate language

code, the program graph, data flow information, the list of required elements for unit testing according to the chosen criterion and the instrumented version of the source code, Figure 4. These information are gathered by a static analysis of the code and the activities involved on this step are classified as the static phase of the tool.

The dynamic phase comprehends the compilation and execution of the instrumented version of the source code. This execution generates the paths traversed through the program, which are used to identify the required elements already covered.

4.2 Reusing Unit Testing Information

The strategy used to implement the *Integration Potential Uses Criteria* is presented in this section. This strategy is based on "information reuse", since it takes advantage of the testing information derived during unit testing to achieve integration testing requirements.

The "information reuse" strategy is based on the idea that an integration testing requirement, such as an interprocedural association or an interprocedural subpath, is represented as a combination of unit testing requirements. Thus, the strategy takes advantage of the previous effort applied on computing unit testing requirements to establish the integration testing requirements.

During the application of unit testing on the program the following set of tasks is performed:

1. Instrument the code;
2. Analyze the code to abstract unit testing requirements;
3. Execute the instrumented code;
4. Evaluate the executed paths to determine satisfied requirements.

When performing integration testing, a new task must be included: 2.1) Combine the unit testing requirements to establish integration testing requirements; and Task 4 must be modified to evaluate against the integration requirements.

Only those unit testing requirements that have interprocedural meaning are used to derive integration requirements; this is the case of requirements involving communication variables and requirements involving the calling node. Figure 5 shows an example of the kind of testing requirements that are of interprocedural interest. Observe on the example that only 8 associations of all 40 from *compress()* are of interest to interprocedural analysis [4]. These associations involve the set of variables {n, lastc} used as actual parameters on the calling statements to *putrep()* on nodes 5 and 11; they must be combined with

[4]Notice that these associations are required by the Potential Uses Criteria [6] and consequently an explicit usage of the defined variable is not required to establish an association.

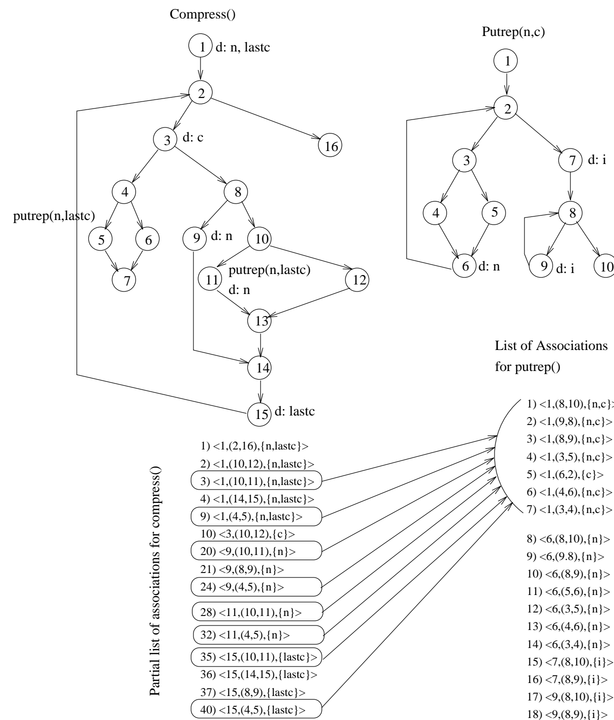

Figure 5: Selecting Integration Testing Requirements.

the set of 7 associations of all 18 from *putrep()*. These associations involve the formal parameters on *putrep()* and the initial node, representing its interface with *compress()*.

A schema representing the approach is shown on Figure 6. With this basic definition a number of different Integration Testing Criteria are implemented, basically by varying the strength with which the combined requirements are created.

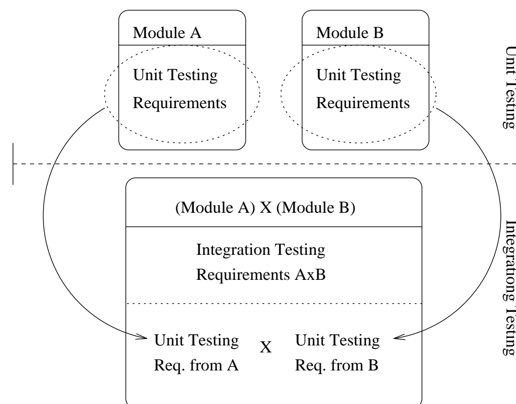

Figure 6: Integration Testing Requirements.

5 Conclusion

A Family of Integration Data Flow Testing Criteria was defined, it is called the Integration Potential Uses Criteria. The criteria are similar to those defined by Linnenkugel and Müllerburg, but differ on the way the interprocedural data flow association is established. Our approach incorporate the concept of potential uses. This small difference promotes a significante difference on the characteristis of the criteria. The analysis of the subsumption relation of these criteria reveled that Integration Potential Uses Criteria includes the *All-Multiple-Relations* criterion therefore enforcing a basic characteristic of testing criteria.

A initial analysis on the implementation of the criteria defined what was called information reuse. Basically it can be described as an attempt to reduce the integration testing cost by taking advantage of unit testing information; the difference is that the unit testing requirements from each unit are now required to be executed in combination with other unit's requirements according to the pairwise approach.

A further study of the applicability of such strategy is still needed. How scalable it is when considering large programs? Empirical studies to define the cost and complexity of the approach are necessary. Testing tools are essential support for undertaking such empirical studies, and the development of the tool to implement the proposed criteria is essential to the evolution of this work.

References

[1] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, New Work, 1990.

[2] M. Chaim. Poke-tool — uma ferramenta para suporte ao teste estrutural de programas baseado em análise de fluxo de dados. Dissertação de Mestrado, DCA/FEE/-UNICAMP, Campinas – SP, Brasil, Abril 1991.

[3] P. Herman. A data flow analysis approach to program testing. *Australian Computer Journal*, 8(3), November 1976.

[4] J. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Trans. Soft. Eng.*, SE-9(3), May 1983.

[5] U. Linnenkugel and M. Müllerburg. Test data selection criteria for (software) integration testing. In *First International Conference on Systems Integration*, pages 709–717, Morristown, New Jersey, April 1990. Systems Integration Conference.

[6] J. Maldonado. Critérios potenciais usos: Uma contribuição ao teste estrutural de software. Tese de Doutorado, DCA-FEE-UNICAMP, Campinas – SP, Brasil, Julho 1991.

[7] J. Maldonado, M. Chaim, and M. Jino. Seleção de casos de testes baseada nos critérios potenciais usos. In *II Simpósio Brasileiro de Engenharia de Software*, pages 24–35, Canela, RS, Brazil, October 1988. In Portuguese.

[8] S. Ntafos. On required element testing. *IEEE Trans. Soft. Eng.*, SE-10(6), November 1984.

[9] R. Pressman. *Software Engineering: A Practitioner's Approach.* McGRAW-HILL, 1992.

[10] S. Rapps and E. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Soft. Eng.*, SE-11(4), April 1985.

[11] H. Ural and B. Yang. A structural test selection criterion. *Information Processing Letters*, 28:157–163, 1988.