Detecting Ripple Effects of Program Modifications on a Software System's Functionality

A Dissertation

Presented to the

Graduate Faculty of the

University of Louisiana at Lafayette

In Partial Fulfillment of the

Requirements for the Degree

Doctor of Philosophy

Jean-Christophe Deprez

Spring 2003

Detecting Ripple Effects of Program Modifications
on a Software System's Functionality

Jean-Christophe Deprez

APPROVED:

Dedicated to my daughter, Emma

Acknowledgements

Deepest thanks to my advisor, Dr. Arun Lakhotia. Since the beginning of my graduate studies in 1995, Dr. Lakhotia has been a very influential person. He has made me a research scientist, and that is invaluable. I also want to give my greatest gratitude to my other committee members, Dr. William R. Edwards Jr. and Dr. Vijay V. Raghavan. Their comments significantly improved this dissertation.

I am also grateful to Dr. Hira Agrawal from Telecordia Technologies for giving me free access to the χSuds™ toolsuite. This greatly helped in the implementation of Sonar. My thanks to Elisha Trombley for her editing work.

Je ne pourrai jamais assez remercier ma femme, Fabienne, et ma mère, Nadine. Je n’y serai jamais arriver sans vous deux. Vos encouragements, vos conseils et votre patience ont fait toute la différence.

Table of Content

List of Tables

List of Figures

1 Introduction

When designing a change to a software system, a programmer reviews the source code and often wonders if changing the source code at a particular spot might lead to an unanticipated ripple effect on the end-user functionality of the system. This concern is warranted because a segment of source code often participates in the implementation of many software functions.[1] In most cases, the sharing of source code between various software functions is neither documented nor evident from the source code. This dissertation addresses such concerns of a programmer by answering the following question:

> *If the source code at a given spot were modified, which software functions would be potentially affected?*

Our work assumes that this question is raised when a change to the source code is being designed and before the change is applied. When designing a change, a programmer may have multiple alternatives. The programmer may ask the above question for each alternative in order to discover the area of the source code with the least ripple effect. As a corollary, our answer to the question above is only a prediction since it does not take into account exactly how the source code will be modified but only where the change might take place. The notion of prediction is emphasized by the term *potentially* in our question. We define *potentially affected* with the following statement.

> A software function *f* is *potentially affected* by a change at a selected spot of the source code if the segment of source code at that spot participates in the implementation of software function *f*.

[1] A *software function* is a task performed by a software system described from an end-user's viewpoint.

1.1 Motivations

If software change requests were not frequent, our research would have limited impact. However, a useful software system rarely stays unchanged. It undergoes continuous modifications to adapt to changes in the needs of the end user, changes in the business environment, and changes in technology. It has been acknowledged that a maintenance programmer spends a significant amount of effort understanding the program being modified [Sommerville 1992]. Furthermore, a software maintenance process such as that of Parikh and Zvegintzov points to the importance of identifying the software functions affected by a particular maintenance before modifying the system [Parikh and Zvegintzov 1983]. Likewise, in the newer *incremental software development* model used by many companies such as Microsoft and by *Extreme Programming* techniques, an entire system is developed in iteration [Cusumano and Selby 1995, Beck 1999]. In this development model, an entire system is built in layers where the implementation of software functions is added one software function at a time until the whole system is built. At each iteration cycle, the programmer modifies the existing source code to insert a new software function. Therefore, the programmer must already identify the ripple effect that the source code changes have on software functions during the initial development of a software application.

Currently, programmers identify the software functions affected by a change in an ad hoc manner. This often leads to overlooking some of the affected software functions. In turn, the resulting errors, if caught on time, lead to reworking the source code and more testing, resulting in lost time. When the errors are not caught before deployment, these errors impact the system's behavior unpredictably, resulting in poor quality. Our technique will enable programmers to identify the *potentially affected* functionality *before* changing the source

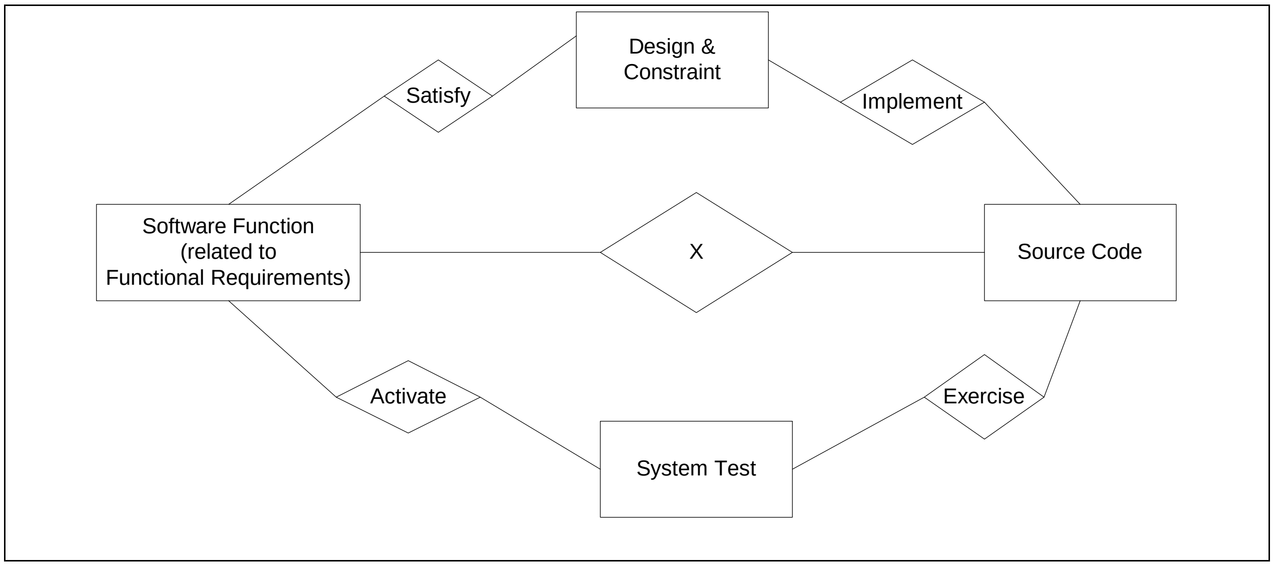


Figure 1: E-R diagram of software.

code. Hence, they will have the direct opportunity to adapt their source code modifications to eliminate undesired side effects. This is expected to improve the quality of modifications made to source code and to reduce the overall time and effort involved in making source code modifications.

1.2 Relating source code to software functions

To answer our opening question, a relation between the software functions of a system and the system's source code must be created. Figure 1 calls this relation *X*. For our purpose, one must be able to use the *X* relationships between source code and software functions to identify the software functions *potentially affected* by a change at a selected spot of the source code.

Figure 1 shows that there are three possibilities to create *X* relationships: (1) directly, (2) transitively using constraints and design, or (3) using system tests. Below, we briefly describe the previous works that have used *X* relationships, and we mention which of the three approaches was used.

Antoniol et al. developed a technique to compute *X* relationships directly [Antoniol et al. 2000]. Their technique computes the probability for a segment of source code to relate to a particular functional requirement based on the similarity of vocabulary used in the requirements documents and in the source code. In other words, a process similar to that used in search engines attempts to match identifiers of the source code to words in the functional requirements document. This technique is highly dependent on how programmers name variables and procedures in a program. As reported in a case study by Antoniol et al., this approach only provides mediocre results when applied on real world systems [Antoniol et al. 2000].

The second way for inferring *X* relationships is by joining information from the *Implement* and *Satisfy* relationships. Commercial companies such as Rational™ and TogetherSoft™ push this approach using design components to infer *X* relationships. Gates et al. also developed a similar approach. Instead of using design components, their approach relates constraints (logic rules) created from the requirements to the source code. In turn, this allows inferring *X* relationships transitively [Gates and Della-Piana 1997, Gates and Li 1998, Gates and Teller 2000]. This approach through design and constraints provides a well-founded framework; however, it requires manual intensive tasks. In fact, not only must the design and constraints be manually created but the *Satisfy* relationships must also be manually created. Moreover, any change to the design, constraints, or requirements document often requires an update of the *Satisfy* relationships. Over time, such an intensive manual effort is likely to introduce errors where the requirements, *Satisfy* relationships, and design become out-of-sync. Hence, the *X* relationships inferred from *Satisfy* relationships may not be reliable. One solution is to automate the current manual maintenance of *Satisfy* relationships. However, this would require very advanced natural language processors and currently may prove too challenging.

A third approach relates software functions to source code by joining information from the *Exercise* and *Activate* relationships. This approach works by observing that the execution of a system test activates software functions and exercises source code. Various efforts have used this approach to locate where requirements are satisfied in the source code implementation [Reps et al. 1997, Wilde and Scully 1995, Wong et al. 1999]. Until the mid-nineties, programmers read the entire traces of source code created by executing system tests, and then they determined what source code segments related to the functional requirement of interest. Thanks to the methods developed by Reps et al., Wilde and Scully, Wong et al., one can directly zoom in to the area of the source code likely to be related to a selected functional requirement. So far, the methods that use *Exercise* and *Activate* relationships provide good information when navigating from software functions to source code. However, nobody has investigated whether *Exercise* and *Activate* relationships enable the inverse navigation from source code position to software functions.

When using *Exercise* and *Activate* relationships for inferring *X* relationships, the main part of the job is to identify a set of system tests needed to achieve the particular goal. All previous works propose techniques to navigate from software functions to source code. In contrast, our goal is to provide a technique for navigating from source code to software functions. Hence, the set of system tests used to achieve the previous goal differs from ours.

In fact, they only require a few system tests to be executed. In contrast, for our purpose, we need to execute many more system tests in order to provide reliable results.

1.3 Measuring the quality of a prediction

Before presenting our objectives, we specify the factors that determine the quality of a prediction: *safety* and *precision*. This will simplify the task of stating our objectives. The two attributes *safety* and *precision,* which determine the quality of a prediction, are independent of the method used to obtain that prediction.

Definition:

- A prediction is *safe* if and only if it identifies all the software functions potentially affected by a change at a selected source code location.
- A prediction is *precise* if and only if all the software functions it identifies are potentially affected by a change at a selected source code location.

In other words, *safety* answers the question "*has our prediction identified all potentially affected software functions?*", and *precision* answers "*Are all potentially affected software functions identified by our prediction?*"

When using the system tests to predict the ripple effect of a source code change on software functions, we know that the set of system tests used for sampling *Exercise* and *Activate* relationships will strongly influence the safety and precision of a prediction.

1.4 Objectives and Challenges

The main objective of our work is to determine criteria for selecting system tests where the resulting *Exercise* and *Activate* relationships predict the software functions potentially affected by a change at a particular source code location with the best possible level of safety and of precision. Secondly, we also want to automate our method for computing predictions as much as possible.

To achieve these goals, we address the following:

1. We must find the adequate techniques for automating the sampling of *Exercise* and *Activate* relationships. **Challenge:** Program profiling helps sample *Exercise* relationships, but we have to develop our own technique for sampling *Activate* relationships. Moreover, several program profiling methods exist; therefore, we must determine the most adequate one for our purpose.
2. We want to identify a set of criteria for system test selection such that the *Exercise* and *Activate* relationships sampled from the execution of these system tests guarantee safe predictions. Moreover, the number of system tests that satisfy these criteria must be finite. **Challenge:** Most software systems accept infinitely many system tests, and the source code implementation of a system often contains infinitely many execution paths. Hence, we must make sure that all needed execution paths in relation to the safety of predictions have been exercised by the execution of a system test. We know that satisfying our criteria for test selection will require a huge set of system tests. Obtaining such set of system tests may not be feasible in practice. Moreover, to guarantee safe predictions, our theory currently does not guarantee the level of precision. These last points lead to our next objective.
3. We want to find criteria for test selection that are satisfied by an acceptable number of system tests. In particular, the number of system tests must be different from the number of software functions of a system by at most a small constant factor. Moreover, the *Exercise* and *Activate* relationships sampled from system tests that satisfy our criteria

must predict the ripple effect of a source code change on software functions with an acceptable, well-determined degree of safety and precision.

1.5 Contributions

Our research makes the following contributions:

1. **A technique for identifying the software functions activated by a system test**. The technique works as follows: Step a) Build a grammar describing the input space of the system; Step b) Annotate each production rule of the grammar with the software functions activated by strings parsed by those rules; Step c) Parse a system test to determine the software functions activated by it.
2. **Sonar, a prototype tool that predicts the ripple effect on software functions by a change at a spot in the source code.**
3. **A system test selection criterion that guarantees safe predictions for a large class of software functions.** This system test selection criterion is based on the notion of interprocedural paths as defined by Melski and Reps. Since an exponential number of tests may be needed to satisfy the criterion, the criterion is not practical. Furthermore, there is no guarantee as to the level of precision of the predictions made using this criterion. In many cases, the resulting precision may be very low. Thus, this criterion is of theoretical significance only.
4. **A second test selection criteria that is practical in the size of system tests needed and the safety and precision of the predictions**. These test selection criteria are satisfied by a number of system tests with a constant relation to the number of software functions. Our case studies on the safety and precision of the predictions based on system tests satisfying this second criterion found that Sonar computed safe predictions

70% of the time and also computed safe and precise predictions between 60–70% of the time for the two systems studied.

1.6 Impacts

Our third contribution states that we found conditions to guarantee safe predictions for a large class of software functions from finite samples of *Exercise* and *Activate* relationships. Currently, it is unpractical to create a set of system tests whose *Exercise* and *Activate* relationships satisfy our condition. However, our finding provides a finite upper bound to the problem of relating a large class of software functions to source code in order to obtain a safe prediction on the ripple effect of a source code change on the software functions. Future efforts may use this bound as a stopping criterion for their algorithms. For example, a small set of system tests would be used to create a few seed *Exercise* and *Activate* relationships. A mechanism would then be used to propagate the seed information to the rest of source code until our bound is reached. Since the bound is finite, we know the propagation algorithm will be tractable.

Part of our fourth contribution is a new set of test selection criteria. These criteria are always satisfied by small sets of system tests. More importantly, the *Exercise* and *Activate* relationships sampled from the execution of these small sets of system tests show an improvement over the automated method proposed by Antoniol et al. However, these new criteria must be refined if they are to be used to create seed *Exercise* and *Activate* relationships. Currently, 70% of the predictions are safe. For good seeds, we would want the percentage of safe prediction in the high nineties.

Currently, our approach to predict potentially affected software functions should not supersede a programmer’s manual analysis. Nevertheless, programmers should definitely complement their results with predictions computed by our method. A side effect of our case study illustrates that our predictions are likely to provide new information to a programmer when his/her manual analysis is likely to be wrong.

Tools that help the software development process, such as those of Rational™ and TogetherSoft™, may benefit from our approach. Currently, these tools predict the software functions potentially affected by a source code change using the relationships design components have with software functions and source code, respectively, called *Satisfy* and *Implements* in Figure 1. *Satisfy* relationships between software functions and design elements are maintained manually; thus, over time errors are likely to occur. Using our approach provides another means to compute the software functions potentially affected by a source code change. Hence, the prediction of the ripple effect of a source code change on software function could be computed both ways, using *Implements* and *Satisfy* relationships and using *Exercise* and *Activate* relationships. A difference in predictions may show that some *Satisfy* relationships are not up-to-date.

On a more general note, the software industry is moving toward object-oriented, component-based software architecture. Programmers using these programming techniques may benefit from our research more than ever. Unlike programs with procedural/functional architecture whose skeletons usually follow the description of the system's software functions they implement, new architectures put the emphasis on objects and relations between objects. This is done by encapsulating all the code related to an object in the same area of a program. In these new architectures, it is very common for one type of object and its methods to be used in the implementation of several software functions of a system. Therefore, modifications to a shared object can possibly affect all the software functions that share it. In a large system, programmers are not always aware of this code sharing. Lack of this kind of awareness may lead to changes in the source code with disastrous effects on the functionality of a system. Our technique communicates this sharing of code to programmers. Thus, our research is potentially more helpful for systems developed using newer programming technologies such as object-oriented programming.

1.7 Outline of this dissertation

In Chapter 2, we describe our method that uses system tests in order to infer relationships between software functions and source code. We first define the three entities of the model: source code, software function, and program input. Then, we explain how program profiling helps sampling *Exercise* relationships, and we present our technique based on annotated grammar for sampling *Activate* relationships. Chapter 2 then explains how our method combined *Exercise* and *Activate* relationships to predict the ripple effect of a change at a selected source code position on software functions. We conclude Chapter 2 with a presentation of Sonar, a prototype tool that implements our method. In Chapter 3, we identify the conditions needed in order to compute safe predictions. In Chapter 4, we present our case studies that determine how well Sonar compute predictions when the sampled *Exercise* and *Apply* relationships are small. Chapter 5 reviews in more detail the related works presented in this introduction. Chapter 6 presents conclusions and plans for our future works.

2 Predicting *Potentially Affected* software functions using system tests

We present a method to predict the software functions potentially affected by a change introduced at a selected position of source code. Before addressing the particularities of our method, we first describe the domains of inputs (source code components) and of outputs (software functions) in Section 2.1.

We then break down the presentation of our method into two steps. Section 2.2 explains how to sample *Exercise* relationships between system tests and source code components and how to sample *Activate* relationships between system tests and software functions. In Section 2.3, we explain how to combine *Exercise* and *Activate* relationships to infer *Potentially Affect* relationships. Our method uses these latter relationships to compute its predictions.

Here are some definitions and notations used throughout this dissertation.

Definition:

- A *set* of elements contains zero or more elements in no particular order and no element is repeated. A *Singleton* is a set with one element.
- $\wp(S)$ denotes the power set of set *S*. It is the set of all subsets of *S* including *S*.
- A *sequence* of elements contains zero or more elements in a specific order, and an element may appear several times within the sequence.
- An *ordered pair* of two elements *a* and *b* denoted $\langle a, b \rangle$ is a sequence of two elements, where *a* is the first element and *b* is the second element.

- A *collection* of elements is a set or a sequence of elements.
- *R: A* $\propto$ *B* defines a relation *R* between two spaces, namely, *A* and *B*. *R* specifies the relationships between elements of *A* and *B*. A relation may be one-to-one, one-to-many, many-to-one, or many-to-many. Set-theoretic notation can be used to define the domain of a relation *R*. Every element of *R* is an ordered pair $\langle A', B' \rangle$, where $A' \subseteq A$ and $B' \subseteq B$.

2.1 Source code components and software functions

2.1.1 Source code components

Our method is to help during a program understanding exercise. Thus, the source code implementation of a system exists. Below, our definitions explain how the source code is divided into components.

Definition:

- *Source code* of a system consists of all files that implement a system and that a programmer is allowed to change.
- A *source code component* is a partition of source code. A *source code component* may be a file, a procedure, a basic block, a statement, or an expression.
- A *basic block* is “a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end” [Aho et al. 1986, page 528].

Usually, source code components do not share pieces of source code with other source code components. In our work, we have partitioned source code into basic blocks. In the case of basic block, source code is not shared between source code components. Given that in our research we only use *source code components* at the level of *basic blocks*, we interchangeably use these two terms.

To enable interprocedural source code analysis, the above definition of basic block is adapted as follows:

- Two basic blocks are added to every procedure definition. The first basic block corresponds to the start of a procedure. This first basic block is sometimes associated to the syntax that specifies the signature of the procedure. The second basic block corresponds to the end of the procedure. It may be associated to the symbol (or reserved word) that indicates the end of a procedure.
- Every procedure call *c* in a procedure *p* generates two basic blocks, one representing the entry from *p* to the target procedure *c* and the other representing the exit back from the target procedure *c* to *p*.

These adaptations enable a precise recording of the basic blocks exercised during the execution of a software system.

2.1.2 Software functions

Software functions are short names given to the functional behaviors of a system. On the same level, we may also describe a software function as a name given to a set of references to portions of the software documents that document a particular functional behavior.

Definition:

- A ***software function*** *f* of a software system is the name given to a task performed by a software system expressed from the end-user’s viewpoint.

This definition is general, as it does not specify the granularity a task must have in order to be considered a software function. The only specificity of the definition is that a task must be specified from an end-user’s viewpoint. Thus, when listing the software functions offered by a system, one is free to enumerate the functional behavior of a system no matter how generic or specific they may be. For example, a software function of a bank automated teller machine (ATM) may be as generic as *perform monetary transaction* or as specific as that expressed in a test scenario such as *attempt to overdraw cash from checking*.

Our definition of *software function*, however, excludes the internal behaviors of the system transparent to the end-user, as well as the nonfunctional behaviors. For example, the behavior *perform lexical analysis* performed by a compiler is transparent to the compiler user; therefore, it is not considered a software function. Our future efforts will work on including these behaviors as a part of our method.

We prefer introducing the new term *software function* rather than using *functional behavior* or *functionality* because the term *function* naturally combines with the verb activate, which we later use to refer to the relationships between software functions and system tests. We also rule out the term *feature* since it is used to refer to nonfunctional characteristics of a system, which our definition currently excludes.

The remainder of this section uses a bank ATM example to show how to materialize and organize *software functions* in a tree. In this particular example, we extract the list of software functions shown in Table 2 from the textual description of the functional requirements of the bank ATM given in Table 1.

Table 1: A list of functional requirements of our bank ATM.

Functional requirements of our bank ATM
1. The ATM must first authenticate the customer by matching the card number and PIN. 2. If the PIN validation fails three straight times the ATM ejects the card. 3. Once the PIN is validated, the ATM must allow the customer to perform one or more of the following operations: • Check balance of checking or savings account tied to current bank card. • Withdraw cash from the checking or savings account tied to current bank card by specifying a sum that is a multiple of $10. • Deposit a check in the checking or savings account tied to current bank card. • Transfer money from an account associated with the bank card to any other account. 4. After a successful operation, the customer must be able to request a receipt. 5. The customer must be able to cancel an operation at any time before it has started being processed. 6. Failure of any operation, beside a failed PIN validation, must generate an error message on the screen that requires the customer's acknowledgement. Once acknowledged, a receipt detailing the failure is printed. Hence, on failure a receipt is always printed.

Table 2: A list of software functions created from requirements of our bank ATM.

f1 Enter PIN	*f11* Abort withdrawal from savings	*f21* Process transfer from savings to checking
f2 Abort PIN	*f12* Process deposit operation	*f22* Abort transfer from savings to checking
f3 Process a balance operation	*f13* Process deposit in checking	*f23* Process transfer from savings to other
f4 Process balance from checking	*f14* Abort deposit in checking	*f24* Abort transfer from savings to other
f5 Process balance from savings	*f15* Process deposit in savings	*f25* Process receipt operation
f6 Abort balance operation	*f16* Abort deposit in savings	*f26* Start another transaction
f7 Process withdrawal operation	*f17* Process transfer from checking to savings	*f27* Process operation on checking
f8 Process withdrawal from checking	*f18* Abort transfer from checking to savings	*f28* Process operation on savings
f9 Abort withdrawal from checking	*f19* Process transfer from checking to other	*f29* Process a money transaction
f10 Process withdrawal from savings	*f20* Abort transfer from checking to other	

Table 2 is a flat list. However, some software functions are not totally different from each other; hence, it is more convenient to classify them in a hierarchy built using a generalization/specialization relationship between software functions. By definition of the generalization/specialization relation, a hierarchy of software functions always holds the following properties:

Property:

- If a software function *f* is a generalization of *f′* then
  - *f* is an ancestor of *f′* in the hierarchy and
  - The activation of *f′* automatically means that *f* (and all other ancestors of *f′*) is also activated.
- A hierarchy of software functions is either a tree or an acyclic graph because a specialization cannot be more general than its ancestors.

Such hierarchical organization facilitates the assessment of a prediction computed by our method. In particular, one can directly know that an entire subtree of software functions is unaffected by simply viewing that the root software function of that subtree is not affected. For example, in our ATM bank, if a prediction shows that a change does not affect the *process withdrawal* software function, then we automatically know that the specialized versions of that software function *process withdrawal from checking* and *process withdrawal from savings* are not affected. When structuring software functions in a hierarchy, this information is directly visible as compared to presenting them in a flat list such as Table 2.

The generalization/specialization relationship between software functions is the basic concept of our classifications; however, there exist different techniques to specify such an organization. In particular, one may use object-oriented, function-oriented, and state-oriented viewpoints to determine whether two software functions are related. In the object-oriented model, one specializes the hierarchy of software functions according to the objects and their attributes. In the case of the function-oriented classification, the focus is on the action performed by software functions. In state-oriented classification, the system functionality is partitioned according to the different states that the system can be in. Some software functions can only be activated when the system is in one particular state and not any other. These three classification techniques are also the most prominent ways of determining suitable organizations of the requirements of a system [Davis 1993, page 21].

Table 3 shows the three types of hierarchies for the list of software functions given in Table 2. Each hierarchy provides a different point of view on the world of the bank ATM's software functions. One may also construct a hybrid hierarchy where more than one of the three classification techniques is used to organize a group of software functions. A resulting hierarchy of software function often has a tree structure, but it may also be an acyclic graph; this is usually the case when organizing software functions using a hybrid hierarchy.

Finally, we define the concept of *complete specialization*. We later use that concept to express an interesting property between software functions and their source code implementation.

Definition: A set F of software functions is a ***complete specialization*** of a software function f if the activation of f also implies the activation of at least one $f_i \in F$.

Table 3: Software functions organized in function-oriented, object-oriented, and state-oriented hierarchies.

Function-oriented hierarchy	Object-oriented hierarchy	State-oriented hierarchy
Bank ATM	**Bank ATM**	Bank ATM
Process	**PIN**	PIN validation
Enter PIN (f1)	Enter PIN (f1)	Enter PIN (f1)
Request Balance (f3)	Abort PIN (f2)	Abort PIN (f2)
from checking (f4)	**Account**	Transaction mode
from savings (f5)	**Checking**	Enquiry transacation mode
Withdraw (f7)	Process balance enquiry (f4)	Balance from checking (f4)
From Checking (f8)	Process withdraw from (f8)	Balance from savings (f5)
From Savings (f10)	Process deposit on (f13)	Abort balance enquiry (f6)
Deposit (f12)	Process transfer from (f17/f19)	Money transaction mode (f29)
From Checking (f13)	Process transfer to (f21)	One account transaction
From Savings (f15)	Abort withdraw from (f9)	Process withdraw from checking (f8)
Transfer	Abort deposit on (f14)	Abort withdraw from checking (f9)
From Checking to savings (f17)	Abort transfer from (f18/f20)	Process withdraw from savings (f10)
From Checking to others (f19)	Abort transfer to (f22)	Abort withdraw from savings (f11)
From Savings to checking (f21)	**Savings**	Process deposit in checking (f13)
From Savings to others (f23)	Same subtree as Checking Except here action are performed on savings.	Aborted deposit in checking (f14)
Print receipt (f25)	**Receipt**	Processed deposit in savings (f15)
Start another transaction (f26)	Print (f25)	Aborted deposit in savings (f16)
Abort	**Transaction**	Two account transaction
Same as subtree as Process except in this subtree actions are aborted.	Start another (f26)	Processed transfer from checking to savings (f17)
		Aborted transfer from checking to savings (f18)
		Process transfer from checking to other (f19)
		Aborted transfer from checking to other (f20)
		Processed transfer from savings to checking (f21)
		Aborted transfer from savings to checking (f22)
		Process transfer from savings to other (f23)
		Aborted transfer from savings to other (f24)
		Print receipt mode
		Print receipt (f25)
		Next transaction mode
		Start another transaction (f26)

As a final note, we observe that there is no direct relationship between software functions and function points. Function points estimate the effort needed to implement a system [Albrecht and Gaffney 1983]. They do so by categorizing and counting the inputs and the outputs of a future system. On the other hand, in our case, software functions are not addressing a future yet unimplemented system, but they are a nomenclature of the functional behaviors of an exiting system. Eventually, we may say that by estimating the effort to implement a system, function points also gauge the effort needed to implement the software functions of a system. However, there is not a quantitative correlation between a chunk of function points and a software function.

2.2 Activate and Exercise: the basic relationships

Our method proposes using system tests to relate source code components to software functions. System testing verifies whether a completely integrated software system conforms to the requirements. Therefore, a system test corresponds to the execution of a particular system test scenario. System testing activity implies two things:

1. Software functions are being tested by system tests. We say that a system test *activates* the software functions being tested; therefore, there exists *Activate* relationships between system tests and software functions.
2. A system test requires the execution of the system. During a test run, it is possible to record the source code components *exercised*. We say that there exist *Exercise* relationships between software tests and source code components.

In section 2.2.1, we explain how program profiling helps in sampling *Exercise* relationships. In Section 2.2.2, we describe a technique to compute some *Activate* relationships.

2.2.1 System tests *exercise* source code components

We first define the *Exercise* relation that exists between system tests and source code components. We then explore how program profiling helps automate the sampling of *Exercise* relationships between system tests and source code components.

Definition: $Exercise : T \propto C$ defines a set of relationships between system tests and source code components. *T* is the space of all potential system tests for a system, and *C* is the set of all the source code components that implement a system.

For most systems, there exist an infinite number of potential system tests; in turn, there exist infinitely many potential *Exercise* relationships. The program profiling technique presented below enables the sampling of a finite number of *Exercise* relationships.

Program profiling consists of recording information about the execution of a software system with a particular test phrase. The information recorded is called an execution profile. An execution profile collects information such as the source code components exercised during a run, the memory usage, the CPU time spent in a particular procedure, etc. For our method, we are only interested in profiling the source code components exercised during an execution. We refer to a collection of the source code components exercised during a particular run as an *exercise trace*.

Different profiling techniques record exercise traces in different formats. The following are the most common profiling techniques and their corresponding exercised trace format:

- *Node* profiling records an exercised trace as a set of source code components.
- *Branch* profiling (or edge profiling) records an exercised trace as a set of ordered pairs $\langle c_1, c_2 \rangle$ where the flow of execution has gone from c_1 to c_2.

- *Path* profiling records an exercise trace as a sequence of source code components. There exist several path-profiling techniques such as intraprocedural or interprocedural path profiling.

An important observation must be made at this time. The current definition of *Exercise* states that every *Exercise* relationship relates a system test to a source code component, not to an exercised trace. Hence, given a set of *Exercise* relationships sampled by profiling the execution of system tests, it is possible to find all the source code components related to a particular system test. However, the sequencing in which source code components were exercised is lost. In other words, given the current definition of the *Exercise* relation, the information available in *Exercise* relationships is as if they were collected using node profiling. The extra sequencing information that branch and path profiling techniques save would actually be lost. The next important fact is that, as we will see in Section 2.3, our method to predict the software function potentially affected does not make use of sequencing between source code components. Thus, no harm is done to the applicability of our method when simply using node profiling.

However, as we will see in Chapter 3, if we want the resulting predictions made by our method to have certain properties, system tests will have to achieve a source code coverage expressed in terms of path. We will present more detail on path profiling in Chapter 3.

We further observe the following about program profiling and its use for sampling *Exercise* relationships. Program profiling collects an exercise trace, but an exercise trace by itself is not an *Exercise* relationship. To sample *Exercise* relationships, a link between a system test and each source code components of an exercised trace must be saved. Thus, we must have a unique way to refer to a system test. This is achieved by assigning a unique name to every system test. Normally, system tests are given unique names as they are executed or as they are specified in the test documentation. Using system test's unique names and their corresponding exercise traces, it is then possible to save *Exercise* relationships.

Let us now illustrate how *Exercise* relationships are created between a system test and the source code components of our bank ATM. First, we present the system test specification. We then show the exercise trace created when the system test is executed. Finally, we list the *Exercise* relationships sampled between the system test and the source code components.

System test with unique name *t* consists of the following interaction: The customer

1. Enters a valid PIN,
2. Withdraws $100 from checking successfully (this implies the customer has more than $100 in his/her account),
3. Requests no receipt, and
4. Does not start another transaction.

Figure 2 shows the exercise trace created when executing system test *t*. When listing the *Exercise* relationships below, we use the notation $X_i..X_j$ to delimit a source code component (or basic block) that starts on line *i* and terminates on line *j* of procedure *X*. When the source code component starts and ends on the same line *i*, we simply denoted it X_i. Using set notation to enumerate the *Exercise* relationship created using the process above, we get

```
Main process of bank ATM
Begin of main process

M1 card_info = readCard();
M2 success = validateProcess(card_info);
M3 if (success = False) then
M4    sendCard();
M5    exit;
M6 endif
M7 cust_rec =
M8   bank_db.getCustomerRecord(card_info);
M9
M10 repeat {
M11
M12   op = doOperationMenu();
M13   // abort then goto next op.
M14   if (op = ABORT) then
M15      goto NextOp;
M16   else // valid op then as for account
M17      acnt = getAccount(SIMPLE_MENU,
M18        cust_rec);
M19      if (acnt = null) then
M20         goto NextOp;
M21      endif
M22   endif
M23   // Withdraw op.
M24   if (op = WITHDRAW) then
M25      from_acnt = acnt;
M26      to_acnt = null;
M27   // Deposit op.
M28   else if (op = DEPOSIT) then
M29      from_acnt = null
M30      to_acnt = acnt
M31   // Balance op.
M32   else if (op = BALANCE) then
M33      from_acnt = acnt;
M34   // Transfer op.
M35   else if (op = TRANSFER) then
M36      from_acnt = acnt;
M37      // for transfer need target account
M38      to_acnt = getAccount(COMPLEX_MENU,
M39        cust_rec);
M40      if (to_acnt = null) then
M41         goto NextOp;
M42      endif
M43   endif
M44   if (op != BALANCE) then // money op.
M45      amount = doAmountMenu();
M46      if (amount = ABORT) then
M47         goto NextOp;
M48      else if (op = WITHDRAW) and
M49         (amount%10 != 0) then
M50         doAmountError();
M51         goto NextOp;
M52      endif
M53      performMoneyTransaction(from_acnt,
M54        to_acnt, op, amount);
M55   else                  // balance op.
M56      bal_str = from_acnt.getInfoStr();
M57      printReceipt(bal_str);
M58   endif
M59
M60    // jump here in case of failure
M61    NextOp:
M62      next = doNextOpMenu();
M63 until (next = False) // end of repeat loop
M64 sendCard();
End // of the main process
```

```
Boolean validationProcess(CardInfo c info)
Begin
P1  success = False;
P2  attempt = 0
P3  repeat {
P4    pin = doPINMenu();
P5    attempt = attempt + 1;
P6    if (pin = ABORT) then
P7       break;
P8    endif
P9    success = c_info.validateCustomer(pin);
P10   if (success = False) then
P11      doPINErrorMenu();
P12   endif
P13 until (success) or (attempt = 3)
P14 return success;
End

Account getAccount(int menu_type,
  CustomerRecord cust_rec)
Begin
A1  acnt_no = doAcntMenu(menu_type);
A2  the_acnt = null; //Assume failure or abort
A3  if (acnt_no = CHECKING) then
A4    msg = cust_rec.getChecking(the_acnt);
A5  else if (acnt_no = SAVINGS) then
A6    msg = cust_rec.getSavings(the_acnt);
A7  else if (acnt_no = OTHER) then
A8    other = doAccountNoMenu();
A9    the_acnt =
A10     bank_bd.getAccountByNumber(other);
A11 endif
A12
A13 // print error message
A14 if (the_acnt = null) then
A15   str = msg.getFormatedString();
A16   printReceipt(str);
A17 endif
A18 return the_acnt;
End

void preformMoneyTransaction(Account from acnt,
Account to_acnt, int op, int amount)
Begin
T1  if (op = WITHDRAW) then
T2    msg = from_acnt.withdraw(amount);
T3    if (msg.noError()) then
T4       sendCash(amount);
T5    endif
T6  else if (op = DEPOSIT) then
T7    msg = to_acnt.deposit(amount);
T8  else if (op = TRANSFER) then
T9    msg = from_acnt.transfertTo
T10     (to_acnt, amount);
T11 endif
T12 if (msg != null) and (msg.error()) then
T13   str = msg.getFormatedStr();
T14   printReceipt(str);
T15 else
T16   // Ask if customer wants receipt
T17   receipt = doReceiptMenu();
T18   if (receipt = YES) then
T19      str = msg.getFormatedStr();
T20      printReceipt(str);
T21   endif
T22 endif
End
```

Figure 2: Highlighted lines of source code are the exercised trace of system test *t*.

$\{\ (t, M_1), (t,M_2), (t,M_3), (t,M_7..M_8), (t,M_{12}), (t,M_{14}), (t,M_{17}..M_{18}), (t,M_{19}, M_{24}), (t,M_{25}..M_{26}), (t,M_{44}, M_{45}), (t,M_{46}), (t,M_{53}..M_{54}), (t,M_{61}), (t,M_{62}), (t,M_{63}), (t,M_{64}), (t,P_1..P_2), (t,P_4), (t,P_5..P_6), (t,P_9), (t,P_{10}), (t,P_{13}), (t,P_{14}), (t,A_1), (t,A_2..A_3), (t,A_4), (t,A_{14}), (t,A_{18}), (t,T_1), (t,T_2..T_3), (t,T_4), (t,T_{12}), (t,T_{17}, (t,T_{18})\ \}$

The enumeration above only specifies the *Exercise* relationships sampled by the execution of system test *t*. When executing many system tests, many more *Exercise* relationships can be collected in the same fashion. However, that sample is never complete since there exist infinitely many potential *Exercise* relationships in an *Exercise* relation. In other words, for any practical purpose, only a finite number of system tests are executed; thus. the *Exercise* relationships sampled never constitute a complete *Exercise* relation.

The *Exercise* relation is many-to-many. In other words, a system test relates to many source code components. Inversely, many system tests may exercise the same source code component.

In the above explanation, we have only considered the case where a system test is a complete execution of the system. Indeed, in our example, system test *t* specifies a list of interactions that corresponds to a complete customer session from entry to exit of the ATM. In other words, a system test is an indivisible unit. Each first element of *Exercise* relationships refers to the unique name of a complete system test. In certain circumstances, it may be desirable to partition a system test into several sequences of interactions, for example when some sequences of interaction are totally unrelated to each other.

Sampling *Exercise* relationships between partial system tests and their corresponding source code components only require a simple adaptation to the profiling technique if a system is not distributed and runs in a single process/thread. In fact, for such a system, the exercise sequence of source code components respects the order of system test interactions. However, in the case of multi-threaded and distributed systems, the change to the profiling technique is nontrivial. For such systems, the exercise sequence of source code components does not automatically follow the order of user interactions. A first and a second series of interactions specified by a system test may exercise source code components concurrently in different processes or different threads. Adding interprocess communication to this scenario makes sampling *Exercise* relationships for a partial system test even more complex. Our intent is to study the applicability of our method with complete system tests. So, we leave changes to the profiling technique for the future.

Let us now briefly mention two techniques that enable performing program profiling. We refer to the first method as *source code instrumentation profiling* and the second as *interpreted profiling*. Source code instrumentation consists of adding code to the source code of a system at compile time. This extra source code assigns a unique identification to each source code component. Subsequently, when the system is executed, the unique identification number of the source code components exercised during a particular run is saved into a file (or database). Source code instrumentation techniques were pioneered by research in source code debugging [Balzer 1969, Hanson 1978, Tolmach and Appel 1990, Agrawal, et al. 1993]. They later found applications in testing, namely test case coverage and regression test selection [Fischer 1977, Fischer, et al. 1981, Harrold and Soffa 1989, Binkley 1995, Rothermel and Harrold 1997, Wong, et al. 1997, Ball 1998, Agrawal 1999]

The interpreted profiling technique only applies to a system that interpreted. An interpreter executes the source code of the system by interpreting it at run time. When instructed to profile execution, the interpreter can additionally collect execution profiles. For example, the java virtual machine (JVM) has a built-in capability for performing profiling. The interface between the profiler and the JVM is defined in the profiling interface JVMPI. This enables a third party to write a profiler that is connected to the JVM at run time.

2.2.2 System tests *activate* software functions

We say that a system test *activates* the software functions being tested.

Definition: $Activate : T \propto F$ defines the relationships between t elements of *T* and of *F*, where *T* is the space of all potential system tests for the software system, and *F* is the space of software functions of the software system.

As for the *Exercise* relation, the *Activate* relation also contains infinitely many relationships between software functions and input phrases. This is derived from the fact that there often exist infinitely many potential system tests for a system.

The *Activate* relation is many-to-many. That is, a system test may, and often does, activate many software functions. Inversely, a particular software function may be activated by many system tests.

Unlike *Exercise* relationships whose sampling must be automated due to the large number of source code components, *Activate* relationships may be collected manually. Indeed, a well-engineered project that follows IEEE 829-1983 Software Documentation Standards directly or indirectly specifies *Activate* relationships [IEEE 1983]. The IEEE 829 standard suggests that test documentation start by the creation of a system test plan at the same time as the requirements analysis phase. The next step is to create a test design specification from which a test case specification is then built. Each test design specifies the features—or software function in our case—of the system the test is addressing. Test cases are then generated for each test design. Hence, when respecting the IEEE 829 standard, relationships between software functions and test cases can easily be extracted from test documentation. When performing a system test, a tester follows the explanations provided by a test case specification; hence, there exists a direct relationship between the actual test and the test case.

Independent of a project respecting the IEEE 829 standard, actual tests will frequently be coded as scripts. A test engine enables running these scripts; as a result, the actual test may be executed automatically. Coding test in scripts is possible irrespective of a system's interface. Test scripts can be created whether the system is command line driven, interactive with text menus, or interactive with a graphical user interface (GUI). Several commercial testing tools such as Rational®Robot by Rational or WinRunner™ by Mercury Interactive Corporation enables the recording of interactions between a tester and a GUI system into scripts.

Over time, companies have accumulated large regression test suites for each of their software applications. In the cases where a large quantity of such test scripts is available for a particular application but where the IEEE 829 standards have not be followed, it is necessary to recover the *Activate* relationships between the test scripts and the software functions of a system in an automated manner. Below we present such a technique.

Recovery of Activate relationships

The recovery method assumes the existence of a series of system tests and of a set of software functions. It analyzes each system test and then determines the software functions it activates. In order to explain our recovery technique, we first specify the information found in a system test.

2.2.2.1 System tests

When IEEE 829 standards are respected, system test documentation found in the test design specification and the test case specification contains all necessary information to determine the software functions activated by a system test. Among others, the pieces of information found in the system test documentation are the following:

- A series of interactions with the system and information on data to be used by the tester.
- A set of input files.
- A set of preconditions that must be satisfied in order to execute the test. These preconditions define what state the system must be in before performing the test. They are specified either in textual descriptions or in formal specifications.
- A list of the expected outputs.
- A set of post-conditions that the state of the system must satisfy after the test.

However, when the IEEE 829 standards on software test documentation are not respected, that information is not available. In many cases, the only system test information available is test scripts and the information needed to run the test script, such as input files referred to by the test scripts or the names of the databases to connect to when running the test scripts.

The fact that only a limited amount of information is available seems limiting. Then again, we have found that many *Activate* relationships can be recovered by only referring to test scripts and their associated input files.

The following example illustrates the system test information available to a recovery technique. We use our bank ATM system for this illustration. As a side note, we imagine there exists a test engine with the ability to run the system test script in Figure 3.

```
Database: “bank1.db”
TestScript: “1234 enter withdraw checking 40 enter no no”
```

Figure 3: Test script named *Withdraw_Checking_success.1.*

First, the caption shows that the system test has the name *Withdraw_checking_success1*. The inside of the system test script contains two lines. The first line specifies the database needed to conduct the system test: *bank1.db*. The second line contains the following list of information: *“1234 enter”* specifies a PIN number needed for authentication, *“withdraw”* that the transaction is a withdrawal, *“checking”* that the transaction is to be performed on the customer’s checking account, and *“40 enter”* that the sum of the transaction is $40. The first *“no”* specifies that no other transaction is to be started, and the second *“no”* specifies that the customer requested no receipt.

2.2.2.2 Recovery techniques

We actually developed two recovery techniques. The first technique has more power but requires the creation of a grammar that expresses the full property of the syntax of test scripts. A grammar is defined by a set of production rules made of terminals and nonterminals from which one is the start non-terminal. We illustrate a grammar for our bank ATM later. The second recovery technique does not require such grammar and, for most software application, retains enough power for the recovery of *Activate* relationships. A description of the first technique appeared in the proceedings of the international workshop on program comprehension 2000 [Deprez and Lakhotia 2000].

For the moment, we focus our attention on recovering *Activate* relationships found in test scripts, and we ignore the information specified next to the test script part of a system test such as the first line of Figure 3, which indicates the name of the database to use when running the script. Afterward, we address the cases where our technique can sometimes use the information found besides the test scripts.

In short, our first technique works in two steps.

1. A grammar able to parse a test script is built.
2. Software functions are specified as parse tree patterns where a parse tree pattern is directly associated to the rules of the grammar.

Thereafter, the software functions activated by a system test can be determined by checking if the parse tree pattern associated with the software function is present in the parse tree of a particular test script. If true, executing the test script activates the particular software function.

Figure 4 illustrates the first step of our technique by giving a grammar that parses the test script of our bank ATM. The grammar is given in Backus Naur Form (BNF). BNF has the power to express context free languages; however, the actual ATM language is regular. We utilize BNF because it is a convenient notation, clearer than its regular expression counterpart. In Figure 4, regular black font represents nonterminals, bold font represents terminals, | means *or*, * means zero or more occurrences, + means one or more occurrences, and ε means empty string.

Grammar for analysis of input space coverage of the ATM:

```
Session ::= Pin Ops
Pin     ::= D D D D Enter | D* Abort
Ops     ::= Transaction No| Transaction Yes Ops | Abort | ε
Transaction ::= Balance Account | MoneyTrans Ticket | Abort
MoneyTrans ::= Transfer Account TransAcnt Amount |
               Withdraw Account Amount |
               Deposit Account Amount |
               Abort
TransAcnt ::= Account | Others AccNum | Abort | ε
Account ::= Checking | Savings | Abort
AccNum  ::= D+ Enter | D* Abort
Amount  ::= D+ Enter | D* Abort | ε
Ticket  ::= Yes | No | ε
D       ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
```

Figure 4: BNF grammar for parsing test scripts of our bank ATM.

Parse tree patterns enable parsing to detect the syntactic features of a test script; therefore, we call the association of a grammar to a partial parse tree *featured grammar*. One can then create a featured grammar for our bank ATM by associating parse tree patterns to the ATM software functions.

In Figure 5, we illustrate parse tree patterns associated with two software functions, namely, *get balance from checking*, and *process withdrawal operation*. The software function *get balance from checking* is represented by the following partial derivation: Transaction ⇒ **Balance** Account ⇒ **Balance Checking**. The software function *processed withdrawal operation* is represented by the following partial derivation: MoneyTrans ⇒

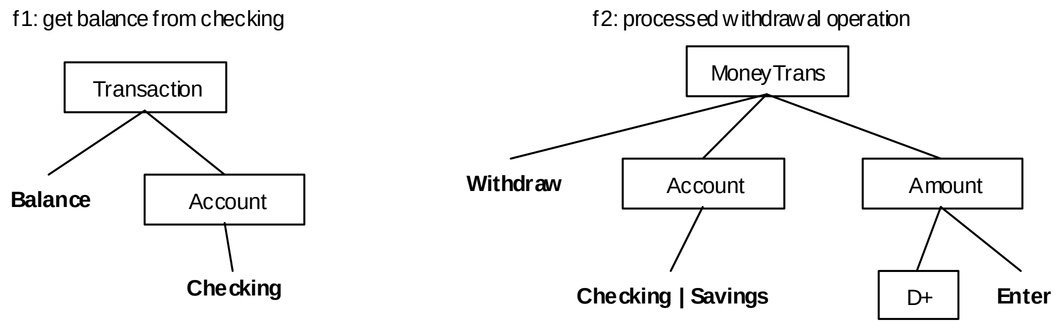


Figure 5: Derivation trees for two features based on the grammar of Figure 8.

Withdraw Account Amount ⇒ **Withdraw (Checking | Savings)** Amount ⇒ **Withdraw (Checking | Savings)** D+ **Enter**.

We now give a formal definition of *featured grammar*.

Definition: FS = (G, F, φ) is a feature syntax of a system where

- *G* denotes a regular or context free grammar that describes the syntax of a particular software application test script,
- *F* denotes the set of software function of the software application, and
- $\varphi: T \propto F$ is a relation that maps a partial derivation tree to a software function.

The φ relation is a many-to-many for the following reasons. In certain cases, a software application implements several ways for the user to activate a particular software function, for example, through menu interaction or using shortcut keys. Therefore, the φ function sometimes maps different partial derivation trees to a software function. Inversely, it is possible that φ maps one partial derivation tree to several software functions. This situation arises when our recovery technique cannot differentiate between two or more software functions. This happens when the difference between software functions is not at the syntax level but at the semantic level. We illustrate such a scenario below.

Let us assume that the list of our bank ATM software functions contains the two software functions *process withdraw from checking* and *overdraw when withdrawing from checking*. The test script given in Figure 3 could activate either one of these two software functions. Unfortunately, our technique cannot determine which software function is activated since it depends on the amount of money in the customer's checking account before running the test script. Of course, the information on the sum of money in an account is available in the bank database, but currently our technique does not make use of data stored in database records. The fact that our technique cannot differentiate between the two software functions is reflected in the many-to-one correspondence from the two software functions to the same parse tree pattern.

Constructing a featured grammar for a particular application is not necessarily a challenging task. However, it introduces work not currently practiced during software development. For real size systems, constructing a complete featured grammar and the relation φ may take significant time. Moreover, adding to the application's user interface requires updating the featured grammar to keep it in-sync with its corresponding software application. Consequently, the software industry will not likely adapt its software development cycle for a technique that requires additional tedious work.

Practitioners are more likely to adopt an approach that builds structures incrementally on-demand. Thus, we propose an alternative method where the construction of a grammar for the complete syntax of test scripts is not needed. Such an approach can be developed by observing the following. The presence of a specific pattern of tokens in the test script of a system test is often sufficient for determining the activated software functions. For instance, if a test script of our bank ATM has the tokens *Balance* and *Checking* in sequence, then the

	Partial Regular Expressions	Software Functions Selected
(a)	‘Checking’	f_{24}
(b)	‘Deposit’ ‘Savings’ [0-9]* ‘Enter’	f_{12}
(c)	‘Withdraw’ (‘Savings’|’Checking’) [0-9]* ‘Enter’	f_{27}
(d)	‘Deposit’ (‘Savings’|’Checking’) [0-9]* ‘Enter’	f_{28}
(e)	‘No’ ‘Yes’	f_{23}

Figure 6: Regular expression patterns of software functions of our bank ATM.

system test activates the software function *get balance from checking*. In such case, simple text matching is sufficient to determine that the software function *get balance from checking* is activated.

In this new technique, one associates software functions with regular expression patterns. These patterns only need to specify a partial regular expression with the few tokens to match in script in order to determine the activation of a software function. That is, the patterns only need to specify regular expressions that parse a part of the test script.

Implementing this second technique is straightforward with engines such as *UNIX egrep*. That is, a programmer associates regular expression patterns to software functions, then, for each pattern, *egrep* identifies the test scripts that match the pattern.

Figure 6 presents some correspondence of function φ between the bank ATM software functions and some partial regular expressions. In our notation of Figure 6, words in single quotes are tokens found in the test script. The other symbols follow the notation of regular expression understood by the UNIX function *regexp*.

Certain software functions need more careful analysis than others. For example, verifying whether a test script activates the *withdraw* function only requires checking for the ‘*Withdraw*’ token. In contrast, verifying if an input phrase applies the software function *print* *a receipt* is more intricate. In this case, we cannot merely specify the '*Yes*' token because the same token is also used to specify the start of another transaction. Thus, in this case, we must specify a partial regular expression that ensures the *'Yes'* token matches the function *request receipt* and not that of *initiate another transaction*. The partial regular expression corresponding to the software function *request receipt* is shown on line (e) of Figure 6.

Partial regular expressions can be developed incrementally on-demand; therefore, they are more likely to be adopted by the software industry. While most scripting languages support regular expressions, writing complex regular expressions is not always easy. For a complex but highly structured input phrase, one may use tree-based regular expressions, such as those provided by *tawk* [Griswold et al. 1996]. Such regular expressions have been successfully used in lightweight techniques for reverse engineering information from programs [Griswold et al. 1996, Ernst et al. 1997].

2.2.2.3 Applicability and limitation of our recovery technique

The language of the ATM is regular; however, the language of other applications, as per the Chomsky hierarchy of languages, may be regular, context-free, context-sensitive, or Turing enumerable. Turing enumerable languages form the largest class. They subsume context-sensitive languages, which subsume context-free languages, which in turn subsume regular languages.

In theory, the language accepted by a software system may be Turing enumerable or context-sensitive. However, the syntax of test script languages for most software application can usually be described using context-free or regular grammar. There exist tools for automatically generating parsers for regular and context-sensitive grammars. These tools may easily be adapted for our purpose. For example, using a platform such as Software

Refinery, one can easily specify a context-free grammar that describes the language of a particular software application's test script. Moreover, the Refine language has the capabilities to specify parse tree pattern using the *rule* construct. Thus, software refinery would be a practical tool for implementing our recovery technique. Other parser generators that allow specifying attribute grammars can also be used for implementing our recovery technique. As mentioned previously, implementing our second technique is straightforward with regular expression matching tools such as *UNIX egrep*.

So far, we have shown how both of our techniques recover the *Activate* relationships between software functions and the test script portion of system tests. However, we have ignored the rest of the information found in a system test, such as input files referred to by a test script or the names and locations of databases used by a test script. That extra information may sometime reveal the activation of additional software functions. Below, we explain how and when our recovery technique can be extended to determine the software functions of information found outside test scripts.

We first give a description of the extension to our technique, and then we give a brief example. The extension associates additional featured grammars to input files requested by test scripts. The content of an input file must be in a known format so a grammar can be created for the particular format. Once the featured grammar for a given format is created, our technique works exactly as for test scripts. In other words, parsing the content of an input file with the associated featured grammar determines the software functions activated by that input file. We illustrate this scenario below.

Let us assume that

- The software application of interest is an HTML viewer;

- A test script specifies the following operation: Open the input file *test.html* in the viewer; and
- The content of *test.html* sets the title of the page in the html header section and specifies a header of level 1 in the body section.

Referring only to the test script, our technique would only identify the activation of software function *open html file*. However, when our technique also uses the appropriate featured grammar associated to the HTML format, it can also determine that the content of *test.html* activates the software function *set title* and the software function *display heading 1*. In this illustration, our second technique, which uses regular expression pattern, can also recover many software functions associated to the HTML syntax.

Unfortunately, creating a featured grammar or regular expression patterns is not always possible for the simple reason that the application programmer does not always know the input file's formats. For example, a programmer does not know the format used by a database management system (DBMS) for storing data. To interact with data in a database, the programmer only needs to know the SQL language. In such cases, our technique cannot analyze the data of the database to determine if they activate software functions.

2.2.2.4 Works related to our recovery technique

Prior efforts used grammars related to the input space of a software application. However, their purpose was not to recover *Activate* relationships but instead to generate input phrases, which could later be used to test the application. In particular, the grammar of a programming language was used to generate programs that were later used to test the compiler [Purdom 1972, Celentano, et al. 1980, Spadafora and Bazzichi 1982, Camuffo, et al. 1990]. These test-generating techniques do not connect grammar to software functions; thus, they are unable to recover *Activate* relationships.

Incidentally, our technique requires much simpler grammars than those used for test phrase generation since we are not worried about the validity of the system test's syntax. In our case, we assume that system tests already exist and are valid. Thus, for our purpose, grammars may overlook certain complexity in the syntax of test scripts (or other input files). It may specify a grammar that parses a superset of the language of test scripts. In contrast, the grammars for test case generation must be precise so as to generate phrases with a perfect syntax structure. Moreover, test generation techniques often require additional methods to validate the semantic of a generated test phrase.

2.3 Potentially Affect

In this section, we explain the term *potentially affect*. In other words, what does it mean for a software function to be potentially affected? We then show that combining *Exercise* and *Activate* relationships infer relationships between source code components and software functions. In turn, these inferred relationships help identify the software functions potentially affected by a change at a specified spot in the source code.

> *A software function f is* ***potentially affected*** *by a change at spot s of the source code if the source code component that contains s* participates to the implementation of *software function f.*

The terminology *participates to the implementation of* is ambiguous. We clarify it by the following assumption.

Implementation participation* assumption:** If a source code component *c* ***participates to the implementation of software function *f*, then there must be a system execution that activates *f* and exercises *c*.

Hence, from the assumption above, we know that

There cannot exist a system test t that activates a software function f and exercises a source code component c where c does not participate to the implementation of f.

The *implementation participation* assumption shows that *Exercise* and *Activate* relationships can be combined to determine the software functions potentially affected by a change at a selected spot of the source code. This will require joining *Exercise* and *Activate* relationships.

The combination of these two types of relationships is done using the standard *select* and *project* operators from relational calculus defined below.

Definition:

- $\sigma_{X'}(R)$ defines the *select* operator. It selects *X'* from *R* where $X' \subseteq X$ and $R{:}X \propto Y$. It returns a set of ordered pairs $\langle X', Y' \rangle$ where $Y' \subseteq Y$.
- $\pi_X(R)$ defines the *project* operator. It projects *R* on *X* where *R* is a relation between domain *X* and some other domain *Y*. It returns a set with the first elements of *R* if *R* is a set of $\langle X', Y' \rangle$ or the second elements of *R* if *R* is a set of $\langle Y', X' \rangle$.

Applying these operators on the *Exercise* and *Activate* relationships sampled from system test allows inferring relationships between exercise traces and software functions. We specify how to infer such relationships below.

Let us assume the following:

1. *T* is a set of system tests, *C* is a set of source code components, and *F* is a set of software functions.
2. The sampling of *Exercise* relationships between system tests in *T* and source code components in *C* has been performed. Similarly, the sampling of *Activate* relationships between system tests in *T* and software functions in *F* has also be performed.

The *Potentially Affect* relationships between source code components and software function are created as follows:

Definition: $PotentiallyAffect = \{\langle \pi_F(\sigma_t(Activate)), c\rangle \mid t \in \pi_T(\sigma_c(Exercise)) \,\&\, c \in C\}$

We can now extract the software functions potentially affected by a change to a spot *s* in the source code.

1. By finding the source code component *c* that contains spot *s*. Spot *s* in the source code is identified by a directory/filename, a line, and a column. With these pieces of information, it is straightforward to find the corresponding source code component *c* that contains a particular spot.
2. By computing the set *F'* of potentially affected software function as follows: $F' = \{\pi_T(\sigma_c(PotentiallyAffect)) \mid c \in C\}$.

A word about the quality of predictions is now in order. Let us first assume that there exists an oracle that always gives a safe and precise prediction as to the set of software functions that will be affected by a change at a specified spot in the source code. Second, let us also assume that for a given spot *s*, the oracle finds that the set F_{pa} is the resulting predictions of the set of software functions potentially affected. We can now define the notion of safety and of precision of a prediction:

1. Safety of a prediction: A set F' of software functions is a safe prediction if $F_{pa} \subseteq F'$. The ratio of safety of a prediction can then be measured as follows. Let $A = F_{pa} - F'$ where $-$ is *set subtraction*; i.e., it removes all elements of F' from F_{pa}. If an element is in F' but no in F_{pa}, then the element is dumped.

   Ratio of Safety = $|A| / |F_{pa}|$.

2. Precision of a prediction: A set F' of a software function is precise if $F' \subseteq F_{pa}$. Alternatively, the ratio of precision of a prediction can be measure as follows. Let $A = F' - F_{pa}$; i.e., remove all elements of F_{pa} from F'. If an element is in F_{pa} but not in F', then the element is dumped.

   Ratio of precision = $(|F_{pa}| - |A|) / |F_{pa}|$.

Thus, *safety* measures how many of the potentially affected software functions are part of a prediction. In contrast, the *precision* measures how many software functions of a prediction are potentially affected. When a prediction is both safe and precise, we say that the prediction is *exact* or *correct*.

Definition: A prediction is ***exact*** (or ***correct***) if it is *safe* and *precise*. Alternatively, we may say that a precision is exact if its ratios of safety and of precision are 100%.

When using our method, a prediction is computed from *Potentially Affect* relationships, which are computed by combining a sample of *Exercise* and *Activate* relationships. In turn, *Exercise* and *Activate* relationships are sampled by a set of system tests. Consequently, the safety and precision of predictions depends on the set of system tests used for sampling the *Exercise* and *Activate* relations.

In Chapter 3, our goal is to find a set of conditions that guarantee safety; however, we do not care about precision. In Chapter 4, we define a set of criteria that a set of system tests must satisfy in order to be used to sample *Exercise* and *Activate* relationships. These criteria do not guarantee safe predictions; however, they improve the precision of predictions.

Besides the level of safety and precision, there is also a *practicality* factor. In our context, practicality characterizes the properties of the set of system tests needed for sampling the *Exercise* and *Activate* relations. If satisfying the properties requires a large number of system tests or if it requires system tests not likely to be in a test suite, then the properties are unpractical. In this research, we do not attempt to measure practicality. We simply state whether a set of specified properties are practical or not. As we will see in our future analysis, the answer on the practicality issue will be obvious.

Before studying our methods, in the next chapters, we present *Sonar*, a prototype tool that uses our method for detecting ripple effects caused by modifying a specified spot of the source code.

2.4 Implementing our method: Sonar

First, we present our design decisions for Sonar and how to prepare the required inputs to use Sonar with a software system. Second, we illustrate applying Sonar with our bank ATM. Then, for demonstrating Sonar at work with our bank ATM, we specify a maintenance task to implement in the ATM, and then we show how Sonar helps during that maintenance task. Finally, we present actual screen shots of Sonar.

2.4.1 Implementing our method

In order to compute predictions, Sonar must refer to *Exercise* and *Activate* relationships. In other words, a preparatory step that samples *Exercise* and *Activate* relationships from system tests is performed before predictions can be computed. In this section, we explain this preparatory step.

Prior to computing and storing *Exercise* and *Activate* relationships, a set of software functions and a set of system tests must be available. We assume that there exists a set *T* of system tests. As indicated in Section 2.2, each system test is held in a file with a unique name, and the file contains the test script plus other information needed to execute the system test. To facilitate the preparatory step, we require a file to list all the unique filenames of the system tests.

Software functions are listed in a *software function definition* file. In addition to the list of software functions, the *software function definition* file also stores the generalization/specialization relationships between the software functions. This allows the software functions to be listed in a tree. Each software function defined in the *software function definition* file also maintains a reference to a *software function activation* file. The *software function activation* file associated to a software function *f* contains a list of the system tests that activate *f*. In other words, the *software function activation* files hold the *Activate* relationships.

The information in a *software function definition* file is as follows:

1. A unique name for the software function being defined,
2. A short and a long description of the software function,
3. The name of the corresponding *software function activation* file, and

4. Two lists of software functions. They enumerate software functions that are specialization and generalization of this software function. These two lists allow structuring software functions into a tree.

In the following, we first address the *Activate* relationships then the *Exercise* relationships.

We know that *software function activation* files hold the *Activate* relationships. The *software function activation* file associated with a software function *f* lists the unique names of the system tests that activate *f*. The two techniques based on *featured grammar* and on *regular expression* pattern could be used to compute the *software function activation* file of each software functions. However, we have left our implementation of Sonar independent of the method used to compute the *Activate* relationships. Sonar requires each *software function activation* file to be associated to a software function in the *software function definition* file. It also requires each *software function activation* file to correctly list the names of the system test that activate the corresponding software function.

Our sampling of *Exercise* relationships uses χAtac, a tool developed by Telcordia that performs node profiling of system runs [χAtac]. χAtac uses source code instrumentation in order to collect exercise traces. It has the ability to instrument source code of the *C* and *C++* languages. By default, χAtac holds all exercise traces profiled in a single *trace* file. However, χAtac allows different names to be given to each exercise trace profiled. This capability gives χAtac the necessary power to hold the *Exercise* relationships needed by Sonar. To obtain a sample of *Exercise* relationships for a set *T* of system tests, the following is done:

1. Compile a software system *S* with χAtac.
2. *For each* system test t in T *do*

a. Let *n* be the unique name of *t*

b. Execute *S* with *t*. Thanks to the special compilation, the profile exercised by *t* is saved in the file *S.trace*.

c. Name *n* the exercise trace just created in *S.trace*

After this step, the file *S.trace* contains the *Exercise* relationships sampled using the set *T* of system tests. χAtac formats *S.trace* so that, given the name of system test, it is simple to extract the set of source code components exercised. However, χAtac does not provide the inverse function, which computes the set of names of system tests that exercised a particular source code component. Sonar needs the latter function to compute predictions. Hence, in our preparatory step, we use the *S.trace* file created by χAtac to precompute the inverse relation and then to cache it.

Sonar is merely a prototype tool. In real life, the capacity of Sonar would likely be integrated in a tool for managing the software development process such as those of Rational™ and Together Software™.

The following is a summary of the list of information needed for preparing Sonar with a particular software system *S*:

- A file that lists a set *T* of system tests with unique names.
- A trace file obtained by executing an instrumented version of the system *S* (instrumented using χAtac) with each system test in *T*.
- A *software function specification* file that defines the software functions and the relationships between them.
- A software function application file for each software function defined in the *software function definition* file.

Our preparatory step precomputes and caches information so that Sonar efficiently computes its predictions for system *S*. After the preparation step, Sonar is ready to compute predictions. To obtain a prediction, the user specifies a filename, a line, and a column. Then Sonar highlights its prediction in the tree of software functions. In the next section, we demonstrate the usefulness of Sonar with an example.

2.4.2 Demonstration of Sonar

First, we illustrate the file needed to prepare our bank ATM system for Sonar. We then demonstrate how a maintenance task on our bank ATM system benefits from the prediction computed by Sonar. Our bank ATM is imaginary so we cannot truly apply Sonar to it. Our demonstration manually computes its prediction exactly as Sonar would. Nevertheless, we conclude this section with actual screen shots of Sonar computing predictions for the software functions of a small spreadsheet application.

2.4.2.1 Preparing our bank ATM

For the purpose of this demonstration, we assume that the ATM is implemented by the source code shown in Figure 2 of Section 2.2.1. The first file required for preparing the bank ATM for Sonar is a file that lists all the system test names (t_i's). Table 4 gives these unique names in the left column. In the right column, we find the test script portion of the system tests.

Now assume that the source code was instrumented with χAtac and that the instrumented system of the ATM was executed with these system tests. This activity creates the second file needed: the *trace* file. Finally, Figure 7 illustrates portions of the *software*

Table 4: List of system tests used for computing *Exercise* and *Activate* relationships for our bank ATM system.

System test names	Test script
t1	12 Abort
t2	1234 Enter Balance Abort
t3	1234 Enter Balance Checking No
t4	1234 Enter Balance Savings No
t5	1234 Enter Withdraw Checking Abort
t6	1234 Enter Withdraw Checking 100 Enter No No
t7	1234 Enter Withdraw Savings 200 Enter Yes No
t8	1234 Enter Deposit Checking 500 Enter Yes No
t9	1234 Enter Deposit Savings 1000 Enter No No
t10	1234 Enter Transfer Checking Savings 500 Enter Yes No
t11	1234 Enter Transfer Checking Other 11122334 123.42 Enter Yes No
t12	1234 Enter Transfer Savings Checking 200 Enter Yes No

```
Process_Mtrans
Process a money transaction
ProcMtrans.html
ProcMtrans.sfa
{ Process }
{ Withdraw, Deposit, Transfer, Checking, Savings }

PWithdraw
Process a withdrawal
PWithdraw.html
PWithdraw.sfa
{ Money_trans }
{ WCheck }

PWChecking
Process Withdraw from checking
PWCheck.html
PWCheck.sfa
{ Withdraw, Checking }
{ }
```

(A)

```
ProcMtrans.sfa:
t6
t7
t8
t9
t10
t11
t12
t13
________________________________
PWithdraw.sfa

t6
t7

________________________________
PWCheck.sfa

t6

________________________________
```

(B)

Figure 7: (A) illustrates a partial software function specification file (only three of many software functions are defined). (B) displays three software function application files. One for each of the three software functions listed in Figure 7(A).

function definition file and the *software function activation* files of the ATM. All the ATM information needed for Sonar is now prepared.

2.4.2.2 Using Sonar during a maintenance

The best method to show the appropriate use of a tool such as Sonar is to propose a maintenance task and then demonstrate when and how the computed predictions assist during maintenance.

Let us assume that the current version of the ATM satisfies the functional requirements given in Section 2.1.2. The team responsible for the ATM realizes that historically most withdrawal transactions are for $20, $40, or $60. Thus, instead of requiring a customer to enter an amount every time a withdrawal operation is selected, the new menu will enable a customer to select a fast cash withdrawal option with the different amounts specified above. The ATM analysts ask a programmer to implement fast cash withdrawal where a customer does not need to enter the amount to withdraw when that amount is $20, $40, or $60. Thus, the functional specification 3 (Table 1) of the current functional specifications changes. It is now:

3. Once the PIN is validated, The ATM must allow the customer to perform one or more of the following operations:
    - Withdraw cash from the checking or savings account tied to current bank card. The withdrawal function must allow a customer:
        - *To select the amount $20, $40, $60 directly without actually typing the amount.*
        - *To enter an amount.*

From this new specification, the programmer knows to study the implementation of the *withdraw* software function and find where the customer is asked to specify an amount. Research by Erdem and Johonson illustrates that to understand a particular behavior, programmers often refer to the exercised traces of input phrases that apply the behavior of interest [Erdem et al. 1998]. In our case, the programmer would refer to the exercise traces of system test that activate the *withdraw* software function.

```
Main process of bank ATM
Begin of main process

   card_info = readCard();
   success = validateProcess(card_info);
   if (success = False) then
      sendCard();
      exit;
   endif
   cust_rec =
     bank_db.getCustomerRecord(card_info);

   repeat {

      op = doOperationMenu();
      // abort then goto next op.
      if (op = ABORT) then
         goto NextOp;
      else // valid op then as for account
         acnt = getAccount(SIMPLE_MENU,
           cust_rec);
         if (acnt = null) then
            goto NextOp;
         endif
      endif
      // Withdraw op.
      if (op = WITHDRAW) then
         from_acnt = acnt;
         to_acnt = null;
      // Deposit op.
      else if (op = DEPOSIT) then
         from_acnt = null
         to_acnt = acnt
      // Balance op.
      else if (op = BALANCE) then
         from_acnt = acnt;
      // Transfer op.
      else if (op = TRANSFER) then
         from_acnt = acnt;
         // for transfer need target account
         to_acnt = getAccount(COMPLEX_MENU,
           cust_rec);
         if (to_acnt = null) then
            goto NextOp;
         endif
      endif
      if (op != BALANCE) then // money op.
         amount = doAmountMenu();
         if (amount = ABORT) then
            goto NextOp;
         else if (op = WITHDRAW) and
            (amount%10 != 0) then
            doAmountError();
            goto NextOp;
         endif
         performMoneyTransaction(from_acnt,
           to_acnt, op, amount);
      else                  // balance op.
         bal_str = from_acnt.getInfoStr();
         printReceipt(bal_str);
      endif

       // jump here in case of failure
       NextOp:
         next = doNextOpMenu();
   until (next = False) // end of repeat loop
   sendCard();
End // of the main process
```

New code could go there

```
Boolean validationProcess(CardInfo c_info)
Begin
   success = False;
   attempt = 0
   repeat {
      pin = doPINMenu();
      attempt = attempt + 1;
      if (pin = ABORT) then
         break;
      endif
      success = c_info.validateCustomer(pin);
      if (success = False) then
         doPINErrorMenu();
      endif
   until (success) or (attempt = 3)
   return success;
End

Account getAccount(int menu_type,
  CustomerRecord cust_rec)
Begin
   acnt_no = doAcntMenu(menu_type);
   the_acnt = null; //Assume failure or abort
   if (acnt_no = CHECKING) then
      msg = cust_rec.getChecking(the_acnt);
   else if (acnt_no = SAVINGS) then
      msg = cust_rec.getSavings(the_acnt);
   else if (acnt_no = OTHER) then
      other = doAccountNoMenu();
      the_acnt =
        bank_bd.getAccountByNumber(other);
   endif

   // print error message
   if (the_acnt = null) then
      str = msg.getFormatedString();
      printReceipt(str);
   endif
   return the_acnt;
End

void preformMoneyTransaction(Account from_acnt,
Account to_acnt, int op, int amount)
Begin
   if (op = WITHDRAW) then
      msg = from_acnt.withdraw(amount);
      if (msg.noError()) then
         sendCash(amount);
      endif
   else if (op = DEPOSIT) then
      msg = to_acnt.deposit(amount);
   else if (op = TRANSFER) then
      msg = from_acnt.transfertTo
        (to_acnt, amount);
   endif
   if (msg != null) and (msg.error()) then
      str = msg.getFormatedStr();
      printReceipt(str);
   else
      // Ask if customer wants receipt
      receipt = doReceiptMenu();
      if (receipt = YES) then
         str = msg.getFormatedStr();
         printReceipt(str);
      endif
   endif
End
```

Figure 8: Exercise trace of system test that activates the software functions *withdrawal from checking* and *withdrawal from savings*.

Figure 8 shows in regular black font the source code components exercised when activating the software function *perform withdrawal from checking* and *perform withdrawal from savings*. In other words, the grayed out code is not exercised by the system tests that activate the two types of withdrawal.

2.4.2.3 Before implementing the software function **fast-cash withdrawal**

When studying the source code components highlighted in Figure 8, a programmer focuses on understanding the implementation of the *withdraw* software function. During that investigation, the programmer realizes that the line of code `”amt=doAmountMenu()”` calls the menu where the customer is asked to enter the amount to withdraw. Thus, a potential solution for implementing *fast-cash withdrawal* is to change this function call. The box *‘New code’* in Figure 8 indicates the spot in the source code where a change could take place in order to implement fast-cash withdrawal.

This is the moment that predictions computed by Sonar are useful. Before designing the source code change, the programmer must know if the solution proposed for implementing fast-cash withdrawal affects software functions other than the withdraw transactions. In other words, when changing the source code at line `”amt=doAmountMenu()”`, what are all the software functions potentially affected?

The programmer does not necessarily have the answer to the question above because during the initial review of the source code, the programmer studied the exercise traces with attention focused on the understanding the *withdraw* software function. During this investigation of the code, the programmer did not necessarily pay attention to finding out the other software functions that could also reach the line of source code of interest.

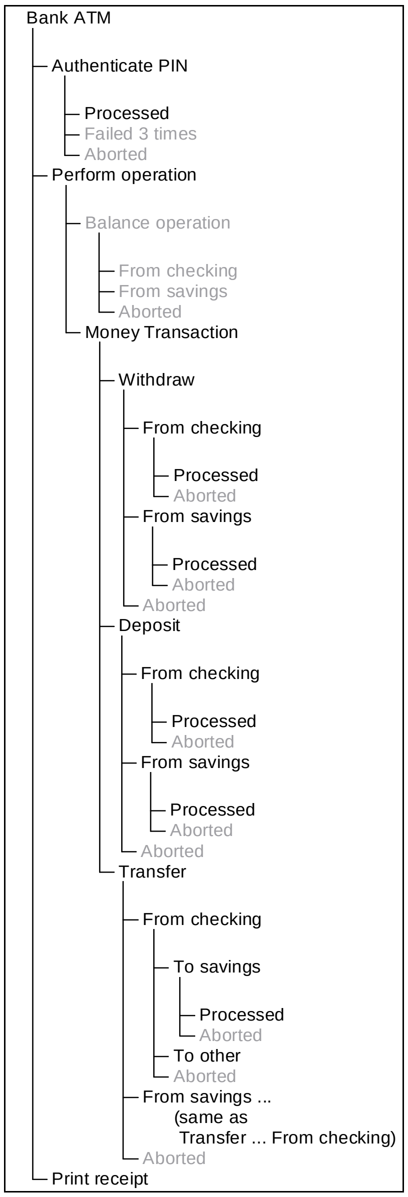


Figure 9: Software functions potentially affected by a change at the position shown in Figure 8.

Using Sonar, the programmer can get an instantaneous prediction that answers the question. Figure 9 shows the software functions of the ATM that would be affected by a change to the line of code "`amt=doAmountMenu()`". The prediction shows that the other monetary transactions, namely, deposit and transfer, could be affected. With that information, the programmer can now design the source code change that modifies the withdrawal software function but does not affect the deposit and transfer transaction. Without the predictions, the programmer might have made a change to the source code that also affected the *deposit* and the *transfer* software function.

We note that the results computed by Sonar also include the *PIN authentication* and the *print receipt* software functions. These two software functions are not directly affected; thus, they could be considered imprecision. Such imprecision is sometimes unavoidable. For example, it is not possible to reach the *withdraw* software function without a positive authentication of a PIN. Thus, Sonar will often predict that the software function *Process PIN Authentication* is potentially affected even when it might not be. On the other hand, other types of imprecision can be avoided. For instance, in our example, the prediction includes the software function *Print receipt* because input phrase *t7* applies *Withdraw* in combination with *Print receipt*. If *t7* did not request a receipt, then the prediction would have been more precise. In Chapter 4, we define criteria for system tests to reduce imprecision.

Finally, one may wonder why the programmer only used the exercise traces of a few *withdraw* operations instead of simply studying the entire code of the ATM. This deeper analysis would have shown the programmer that the deposit and transfer transactions could be affected by a change to the proposed line of source code. In fact, when modifying a small program, studying its entire implementation is the best method. However, when the source code implementation is larger than just a few thousand lines of source code, studying the entire source code is often not a practical option.

We now present a few actual screen shots of Sonar. These screen shots come from the study of the software application scalc, a small spreadsheet program used later in our case study. Although Sonar is mainly built to predict the software functions affected, we have

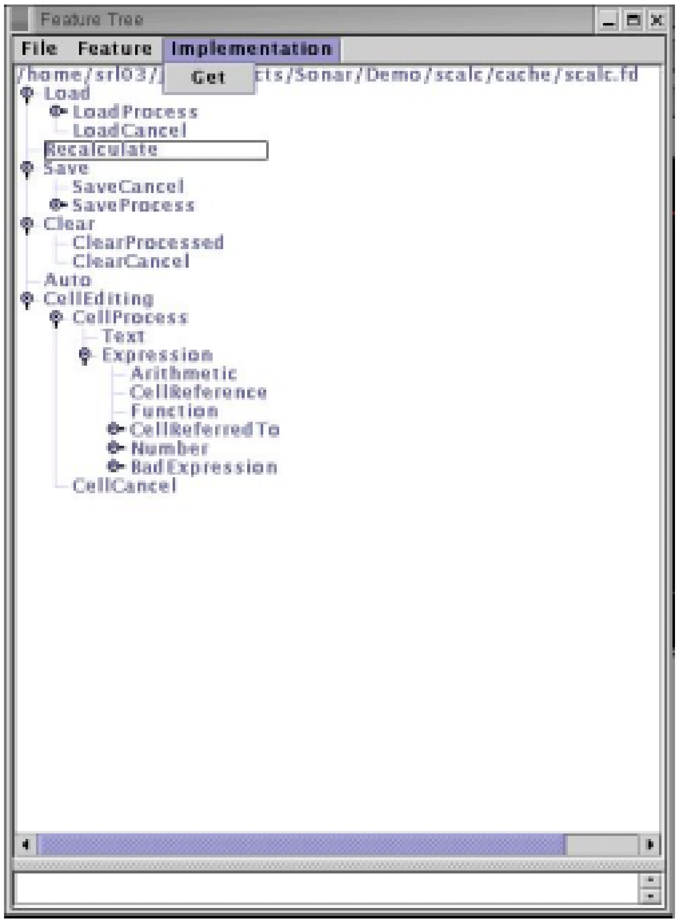
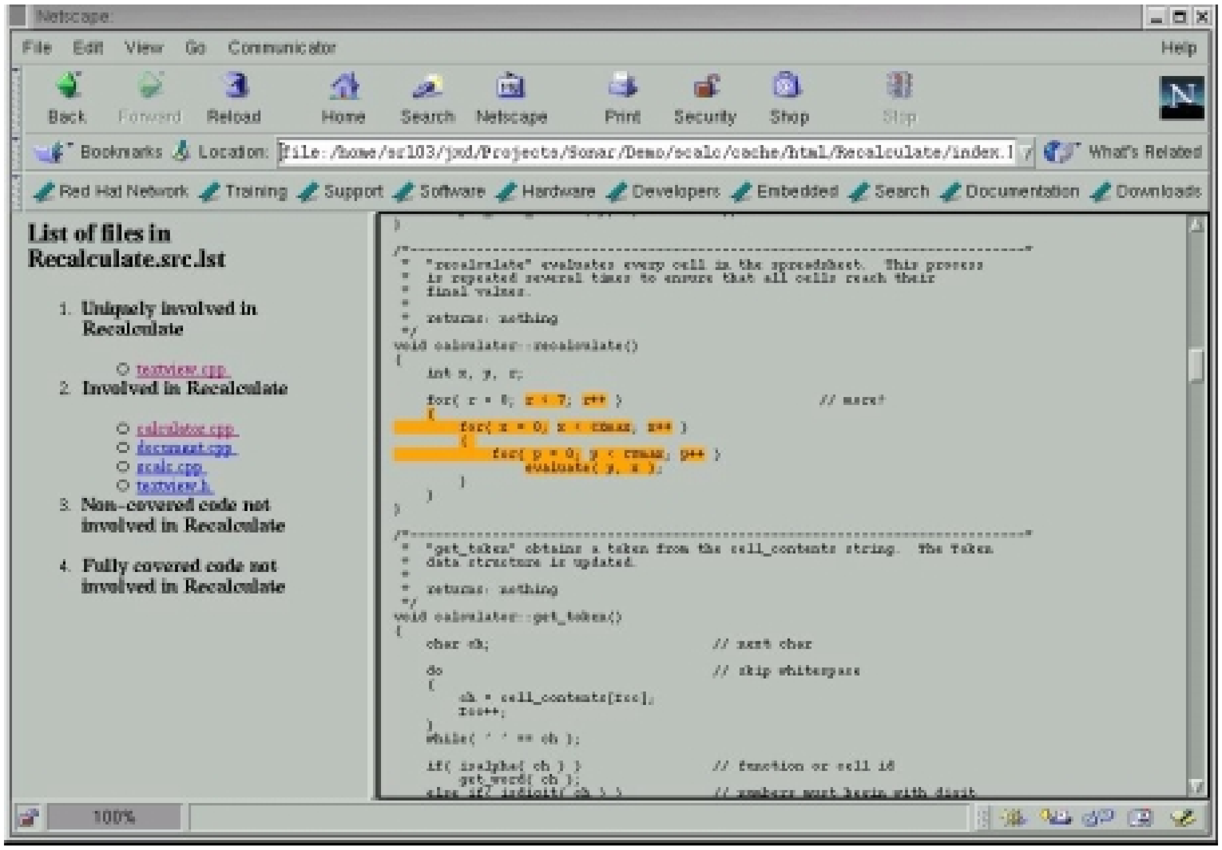


Figure 10: Sonar analyzing scalc.

built it in such a way that it can also project the inverse information, i.e., Sonar can also identify the source code components related to a particular software function. The capacity to project information from software functions to source code components is also found in χSuds from Telcordia.

Figure 10 shows the tree of software functions of *scalc*. This tree is generated from a *software function specification* file. This figure shows that the software function *Recalculate* is selected to identify the source code components that implement the software functions. Sonar highlights the relevant source code using html tags that color the relevant source code. The answer can then be displayed in a web browser such as the Netscape™ web browser (Figure 11).


Netscape:
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: file:/home/sr103/jxd/Projects/Sonar/Demo/scalc/cache/html/Recalculate/index.1
What's Related
Red Hat Network Training Support Software Hardware Developers Embedded Search Documentation Downloads
100%

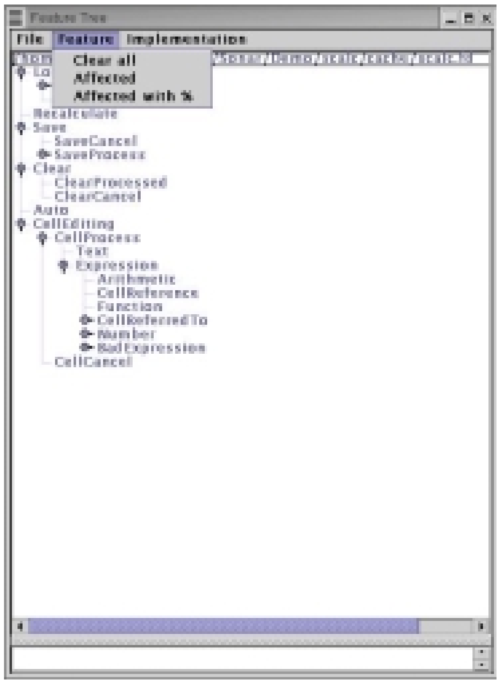
List of files in Recalculate.src.lst

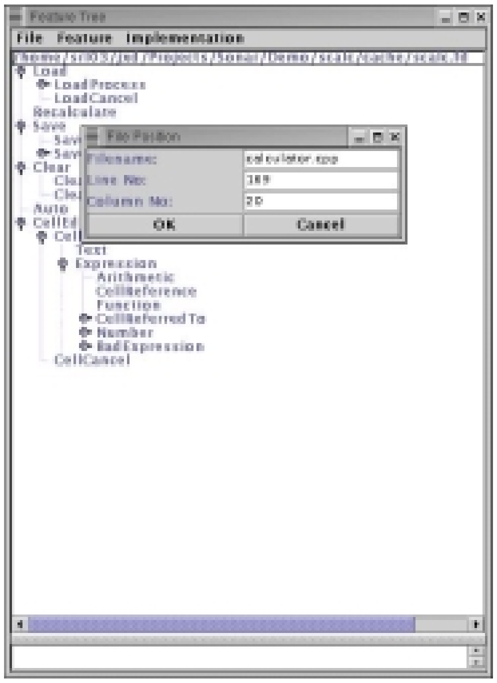
1. **Uniquely involved in Recalculate**
   - textview.cpp
2. **Involved in Recalculate**
   - calculator.cpp
   - document.cpp
   - scalc.cpp
   - textview.h
3. **Non-covered code not involved in Recalculate**
4. **Fully covered code not involved in Recalculate**

```
}

/*---------------------------------------------------------------------------*
 *  "recalculate" evaluates every cell in the spreadsheet.  This process
 *  is repeated several times to ensure that all cells reach their
 *  final values.
 *
 *  returns: nothing
 */
void calculator::recalculate()
{
    int x, y, r;

    for( r = 0; r < 7; r++ )                          // magic
    {
        for( x = 0; x < cmax; x++ )
        {
            for( y = 0; y < rmax; y++ )
                evaluate( y, x );
        }
    }
}

/*---------------------------------------------------------------------------*
 *  "get_token" obtains a token from the cell_contents string.  The Token
 *  data structure is updated.
 *
 *  returns: nothing
 */
void calculator::get_token()
{
    char ch;                                  // next char

    do                                        // skip whitespace
    {
        ch = cell_contents[rcc];
        rcc++;
    }
    while( ' ' == ch );

    if( isalpha( ch ) )                       // function or cell id
        get_word( ch );
    else if( isdigit( ch ) )                  // numbers must begin with digit
```

Figure 11: Highlighted source code is involved in the implementation of software function Recalculate.

We will now illustrate the true function of Sonar: the ability to predict the software functions affected by a change at a particular spot in the source code. In this example illustrated in Figure 12, the spot at line 169 and column 20 of file `calculator.cpp` points to the source code statement `evaluate(y,x)` highlighted in orange in Figure 11. Figure 12 illustrates how a programmer asks Sonar for a predication, and Figure 13 shows the method used by Sonar for presenting its prediction.





Filename: calculator.cpp
Line No: 169
Column No: 20

Figure 12:A programmer wants Sonar to project the software functions affected if source code of `calculator.cpp` at line 169 and column 20 were modified.

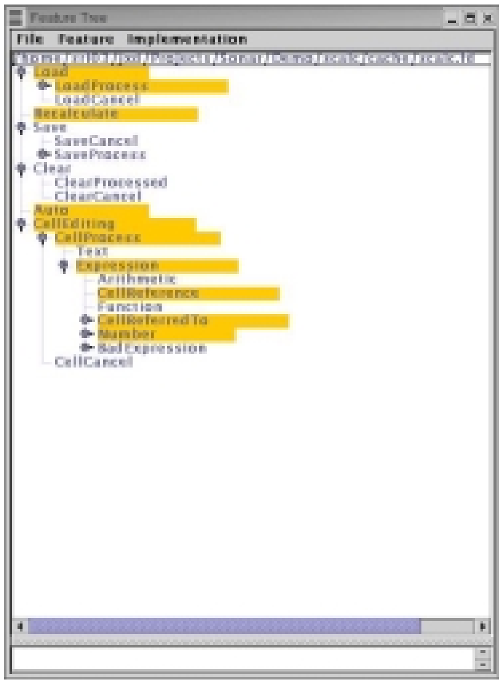


Figure 13: Projection computed by Sonar. Potentiallyaffected software functions are highlighted.

3 Computing safe predictions

In this chapter, our goal is to identify conditions under which our method *safely* predicts the software functions potentially affected by a change at a selected spot of the source code. In Section 3.1, we show how computing safe predictions relates to the method presented in the previous chapter. In turn, this allows our goal to be expressed in terms of coverage of software functions and coverage of source code that a set of system tests must achieve. Section 3.2 shows that using only coverage information is not sufficient to guarantee safe predictions. Therefore, we slightly redefine our goal in Section 3.3. We then propose in Section 3.4 a solution to this new goal where safe predictions are guaranteed for a well-defined, broad category of software functions. Finally, in Section 3.5, we assess our solution.

3.1 Expressing safe predictions with coverage conditions

In the following, we use our definitions and assumptions to connect the notion of safe predictions to our method that computes predictions. This allows us to express our goal precisely. We start from the definition of safe prediction:

A set F' of software functions is a ***safe*** *prediction if $F_{pa} \subseteq F'$*

where F_{pa} is the correct prediction of the software functions potentially affected.

In other words, a prediction is *safe* if it contains all the software functions potentially affected by a change at a specified spot of the source code. To further our analysis, we need the definition of *potentially affected*. We refer to the definition of *potentially affected* given in Chapter 1; however, we replace the phrase *segment of source code* by *source code component* since we have defined the latter.

A software function f is ***potentially affected*** *by a change at a selected spot of the source code if the source code component that contains the spot participates to the implementation of software function f.*

Finally, our method relies on our *implementation participation assumption*. Thanks to this assumption, we can connect the notion of safety to the predictions computed by our method. The assumption states

If a source code component c ***participates in the implementation of*** *software function f, there must be a system execution that activates f and exercises c.*

On the one hand, the *implementation participation assumption* relates *safety* of predictions to the *activation* of software functions and the *exercise* of source code components, and on the other hand, our method uses system tests to sample *Exercise* and *Activate* relationships. So, we initially state our goal as follows.

We want to identify conditions needed by a set of system tests such that

- There always exists a finite set T of system tests that satisfies the condition below.
- When a set T is used to sample Exercise and Activate relationships, our method computes safe predictions.

Conditions on a set of system tests are specified in terms of coverage of software functions and coverage of source code. Thus, we can further refine the way we express our goal:

We want to identify a criterion X of source code coverage and a criterion Y of software function coverage such that

- There exists a finite set T of system tests whose execution satisfies coverage criteria X and Y.
- When set T is used to sample Exercise and Activate relationships, our method computes safe predictions.

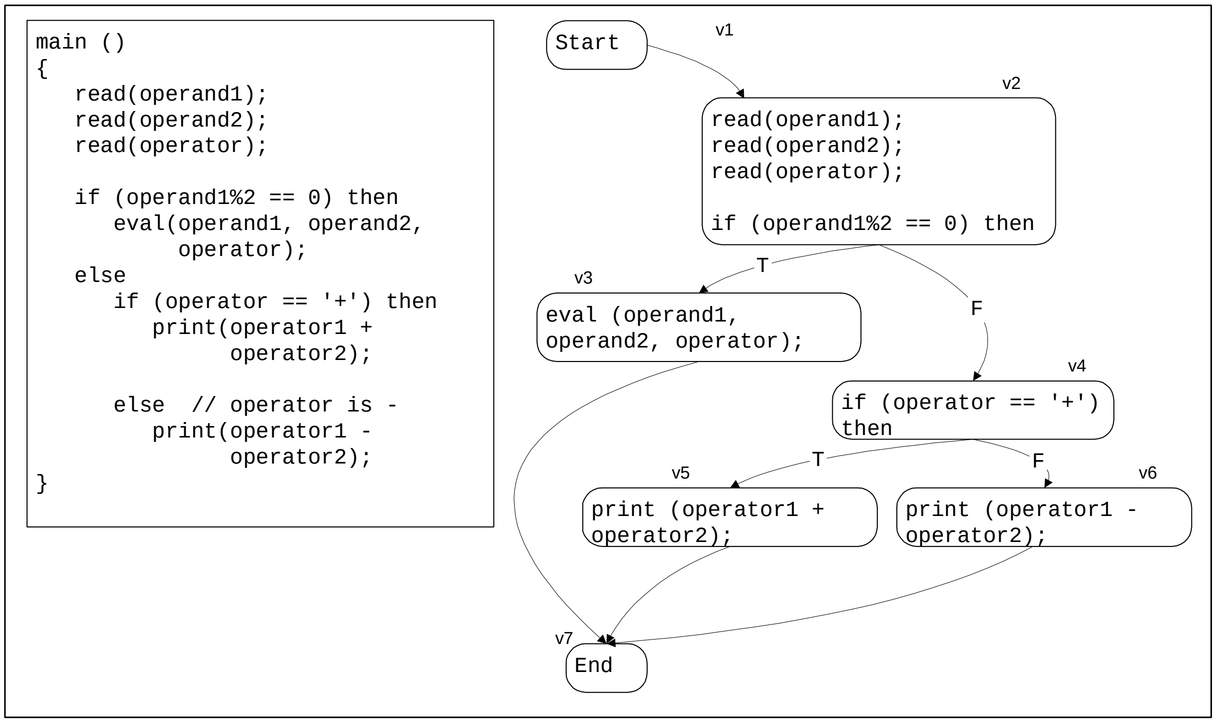
3.2 Predictions based on coverage conditions: unsafe

Unfortunately, when only using information about software function coverage and source code coverage, it is impossible to guarantee safe predictions. No coverage of source code and of software functions is sufficient for guaranteeing that our method always computes safe predictions.

Before illustrating our problem with an example, we first present the two factors that cause the problem:

- *Factor 1*. A software function is completely re-implemented in several areas of the source code. This happens when the implementation of a new software function cannot easily fit in the current source code of a system. In such cases, the software function is implemented several times in different areas of the source code that relate to that new software functions.
- *Factor 2*. The activation of different software functions results in the same path of source code being exercised. This situation can occur when the complete source code implementation of a system is not accessible, for example, when third party libraries in a compiled form are used to implement a system.

A system that combines these two factors in a certain way makes it impossible to guarantee safe predictions. Our example is based on a very simple calculator. In particular, the calculator has only two software functions, namely, *add* two numbers and *subtract* two



```
main ()
{
   read(operand1);
   read(operand2);
   read(operator);

   if (operand1%2 == 0) then
      eval(operand1, operand2,
           operator);
   else
      if (operator == '+') then
         print(operator1 +
               operator2);

      else  // operator is -
         print(operator1 -
               operator2);
}
```

Start
v1
v2
read(operand1);
read(operand2);
read(operator);
if (operand1%2 == 0) then
T
F
v3
eval (operand1, operand2, operator);
v4
if (operator == '+') then
T
F
v5
print (operator1 + operator2);
v6
print (operator1 - operator2);
v7
End

Figure 14: Sample source code and CFG of system used for illustration of unsafe predictions.

numbers. The source code implementation of our calculator and its corresponding control flow graph are shown in Figure 14. For the purpose of our example, let us assume that the procedure `eval` in Figure 14 belongs to a third library and its source code is not accessible.

The control flow graph (CFG) of Figure 14 contains three complete paths from start to end:

- $p_1 = v_1, v_2, v_3, v_7$
- $p_2 = v_1, v_2, v_4, v_5, v_7$
- $p_3 = v_1, v_2, v_4, v_6, v_7$

Let us now point out the presence of the two factors. Factor 1 is present since both *add* and *subtract* have their implementation duplicated in the source code. *Add* is implemented by paths p_1 and p_2, and *subtract* by path p_1 and p_3. Factor 2 is also present since path p_1 implements several software function, in this case the two software functions *add* and *subtract*.

We now present three system tests that respectively exercise the paths p_1, p_2, and p_3. Moreover, the set of these three system tests activates the two software functions *add* and *subtract*.

- t_1 = `2 3 +` activate software function add
- t_2 = `3 2 +` activate software function add
- t_3 = `3 2 -` activate software function subtract

We now show that despite the full coverage of software functions and of the full path of source code achieved by these three system tests, our method still computes unsafe predictions. In particular, let our method compute a prediction for a spot in source code component v_3. Our method finds that

1. t_1 is the only system test that exercised v_3, and
2. t_1 activates the software function *add*.

Our method predicts that a modification to a spot of v_3 potentially affects the software function *add*. This prediction is unsafe since the execution of the following system test

t_4 = `2 3 -` activates software function *subtract.*

Moreover, t_4 also exercises path p_1, which contains vertex v_3. Thus, a safe predictions must include both the software functions *add* and *subtract*. However, referring only to coverage of software function and of source code, we have no way to know that the system test t_4 is needed to guarantee safe predictions. In fact, the execution of the three system tests, t_1, t_2, and t_3, already achieves a full coverage of software functions and of paths of source code.

In conclusion, in the general case, it is not possible to guarantee safe predictions only referring to the coverage of software function and the coverage of path of source code.

There are two ways to remedy this problem. A first solution is broadening our analysis by incorporating semantic checks whose role would be to discover that, for example, the three system tests t_1, t_2, and t_3 of our calculator example are not enough to guarantee safe predictions. Building the required semantic checks would render the application of our method very tedious, given that this problem does not occur frequently in practice.

Thus, for the moment, we prefer a second strategy where we impose a restriction on software functions.

3.3 Expressing safe predictions with restriction

We accept the fact that safe predictions for all software functions cannot always be computed from coverage information only. Instead, we slightly shift our goal by wondering whether there is a well-defined category of software functions for which we can guarantee safe predictions only using coverage information. Obviously, we want this well-defined category of software functions to be as broad as possible.

From this strategy, we can reformulate our new goal as follows.

NEW GOAL:

We want to identify a **restriction Z** *on software functions, a* **criterion X** *of source code coverage, and a* **criterion Y** *of software function coverage such that*

> ***1****. There always exists a finite set T of system tests whose execution satisfies criterion X and criterion Y where criterion Y applies to all software functions that respect restriction Z.*

2. ***IF (****Exercise and Activate relationships are sampled with set T of*
system tests that satisfies
criterion X of source code coverage ***AND***
*criteria Y of software functions coverage****)***
THEN *our method computes safe predictions*
for all software functions that respect restriction Z.

In the next section, we propose a solution to this new goal and show that the proposed solution does in fact fulfill our new goal.

3.4 Computing safe predictions with restriction

In Section 3.4.1, we propose a first attempt where we specify a restriction Z on software functions. This trial fails. However, it teaches important lessons for our next attempt. In Section 3.4.2, we then develop our new solution and prove that this second trial fulfills our goal.

3.4.1 Restrictions on software functions: a first attempt

In this first attempt, we start by proposing a restriction Z on software functions. However, this restriction is not good enough to guarantee safe predictions. For this first effort, we construct restriction Z to avoid the problem mentioned in Factor 2 of Section 3.2, which points out that different system tests can activate different software functions but exercise the same path of source code.

Restriction Z (Attempt 1): System tests that activate different software functions never exercise the same path of source code.

Although this restriction solves the problem raised in Factor 2, it still does not allow guaranteeing safe predictions. In fact, for most software systems, there are infinitely many paths in the source code, and restriction Z does not allow determining the finite set of these paths that must be covered for guaranteeing safe predictions. To clearly point out the problem, we examine a particular case.

Let us assume that a change is to take place at a spot of source code component *c*. Because of loop and recursion, the source code implementation of most systems has infinitely many paths that contain *c*. Currently, restriction Z does not allow determining the finite set of paths needed to obtain a safe prediction for *c*. Randomly selecting a finite number of paths that contains *c* jeopardize the safety of predictions, and exercising all these paths requires an infinite number of system tests.

Consequently, our first attempt fails. However, we draw lessons from it.

Lesson 1. Restriction Z restricts software functions in terms of path of source code. Doing so makes restriction Z have an effect on criteria X and Y. In particular, let us define criterion X as *every path of source code to be exercised by at least one system test*, and let us define criterion Y as *every software function to be activated by at least one system test*. We know that if a set *T* of system tests satisfies criterion X then T also satisfies criterion Y for all software functions that respect Z. Specifying a restriction Z with such a property is useful because we then only need to focus our attention on finding an adequate criterion X.

Lesson 2. Currently, our restriction Z on software functions is not restrictive enough. In fact, to guarantee a safe prediction for a set of software functions that respect restriction Z, an infinite number of paths must be exercised. One strategy for solving this problem is to find a way of connecting restriction Z to a criterion X that is reachable by a finite number of system tests.

In conclusion, from lesson 1, we decide that our restriction Z on software function must guarantee that when criterion X of source code coverage is satisfied, criterion Y of software function coverage is also satisfied for all software functions that respect restriction Z. From lesson 2, we know that restriction Z on software functions must be connected to a criterion X that is reachable by a finite number of system tests.

3.4.2 Our solution for computing safe predictions with restriction

Our lessons learned make a crucial point: restriction Z on software function and criterion X of source code coverage must relate in some way. To achieve a connection between X and Z, we find it practical to first search for units of source code used for measuring source code coverage. Second, we determine how these units of source code can be used to specify a restriction Z on software functions. Third, we use these units of source code to specify a criterion X of source code coverage reachable by a finite set of system tests. Criterion X and restriction Z will be connected since they are specified in terms of the same units of source code. Moreover, these units of source code also allow a restriction Z on software functions to be specified such that when criterion X of source code coverage is satisfied, criterion Y of software function coverage is also satisfied for all software functions that respect restriction Z.

Consequently, to reach our goal, we perform the following steps:

1. Let criterion Y of software functions coverage be:

   Every software function must be activated by at least one system test.

2. Identify different units of source code used for measuring source code coverage in the remaining of Section 3.4.2.

3. First, find how the units of source code help specify a restriction Z on software functions. Then, determine a criterion X of source code coverage that is reachable by a finite set of system tests (Section 3.4.3).

4. Show that we have identified criteria X and Y, and a restriction Z that fulfill the two points of our goal (Section 3.4.4).

Units of source code

Chapter 2 showed how program profiling helps record the source code exercised. Moreover, it presented node and branch profiling. These two types of profiling define two units of source code, namely, nodes (source code components) and branch (ordered pair of source code components). In this section, we go one step further by presenting units of source code that represent paths of source code. Units of source code paths will enable specifying a weaker restriction Z. Since our goal is to find a restriction Z that allows for a broad category of software functions, paths will define better units of source code than node and branches.

First, we propose to define a unit of source code as a full path.

Definition: A ***full path*** is a single sequence of source code components that corresponds to a possible chronological order of exercise during a system test.

Full paths are not convenient because often there are infinitely many full paths in the source code. Hence, it will be the job of restriction Z to determine the finite set of full paths that must be covered in order to guarantee safety. However, all full paths are different, and there are no clear good criteria for determining whether a full path must be selected. In turn, we look at other alternatives.

Next, we present the notion of intraprocedural acyclic path and interprocedural path. Research by Ball and Larus and by Melski and Reps have respectively defined intra- and interprocedural paths and methods for profiling these paths efficiently [Ball and Larus 1996,

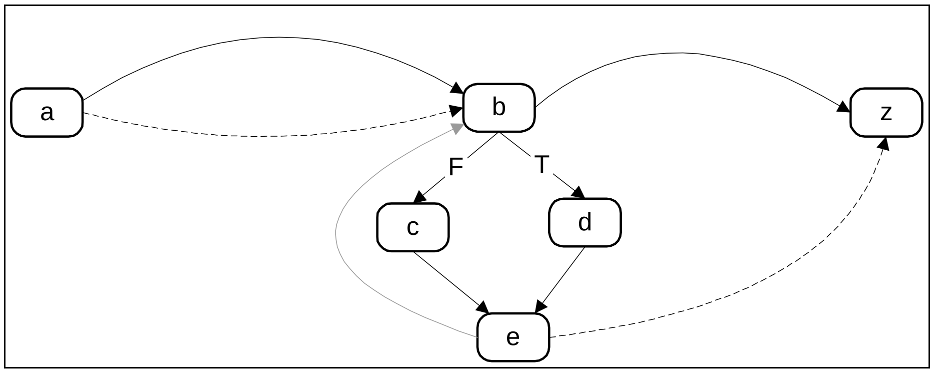


Figure 15: Control flow graph transformed for computing B-L paths.

Melski and reps 1998]. Both types of paths are defined on the control flow graph of the source code.

Definition: A control flow graph *G= (V, E)* is defined by a set *V* of vertices where each vertex $v\in V$ represent a basic block of the source code, and a set *E* of edges. An edge $e\in E$ is represented by an ordered pair $<v_1, v_2>$ where $v_1, v_2\in V$. It indicates that there exists a flow of control from basic block v_1 to basic block v_2. In addition, *V* is augmented with a *Start* vertex and an *End* vertex, and *E* is augmented with an edge <*Start*, *v*> where *v* corresponds to the basic blocks of that is exercised first, and a set of edges <*v*, *Exit*> where each *v* corresponds to a basic block that may be executed last.

Ball-Larus intraprocedural paths

When a procedure contains a loop, there are infinitely many intraprocedural paths in a control flow graph. Ball and Larus propose to summarize the infinite number of paths with a finite number of acyclic intraprocedural paths (B-L paths). Every intraprocedural path can be expressed as a composition of B-L paths.

The definition of B-L path uses the notion of back edges in the control flow graph.

Definition: ***Back edge*** in a directed graph is an edge from basic block *a* to basic block *b* where *b* is always executed before *a*.

In Figure 15, *a* and *z* respectively represent the *Start* and the *End* vertices of the CFG. The gray edge <*e,b*> is a back edge; it is part of the CFG. However, we observe that the dash edges are not. Their use is explained later.

There exist four types of B-L paths:

- Acyclic paths from *Start* to *Exit*, for example, *abz*.
- Acyclic paths from *Start* to a CFG vertex *x* finishing by the execution of $\langle x, h \rangle$ where *h* is the target of a back edge. For example, the sequence made of the single vertex *a* finishing by the execution of $\langle a, b \rangle$.
- Acyclic paths from a CFG vertex *h* that is the target of a back edge to *Exit*, for example, *bz*.
- Acyclic paths from a CFG vertex *v* that is the target of a back edge to a CFG vertex *x* such that acyclic paths finish by executing $\langle x, h \rangle$ where *h* is the target of a back edge. We note that *v* and *h* may be the same node in the case where there are several paths within a single loop, for example, *bce* and *bde*. Both finish by executing $\langle e, b \rangle$

Ball and Larus give a method that computes B-L paths by substituting each back edge (gray edge of Figure 15), with two surrogate edges (dash edges of Figure 15). The first surrogate edge goes from the CFG *Start* vertex to the target of the back edges, and the second edge goes from the source of a back edge to the CFG *End* vertex. Since this transformation removes the intraprocedural cycle, the transformed CFG is acyclic. In turn, there exist a finite number of B-L paths within a procedure. In addition to the transformations, Ball and Larus also provide an algorithm to instrument source code so it records the B-L paths exercised during an execution.

Melski-Reps interprocedural paths

Influenced by Ball and Larus, Melski and Reps defined the notion of interprocedural path. We refer to such interprocedural paths as *M-R paths*. M-R paths are defined on the interprocedural control flow graph (ICFG). We first expand our definition of CFG to define ICFG.

An interprocedural control flow graph (ICFG) consists of a *Global Start* vertex, a *Global End* vertex, and a set of control flow graphs (CFGs), one for each procedure of the source code. As specified in the next section, each CFG has a unique *Start* vertex and a unique *End* vertex. Each vertex in a CFG represents a source code component at the basic block level, except for procedure calls where each call defines two vertices, an *Entry* vertex and an *Exit* vertex. In the ICFG, for every call to a procedure *p*, there is an entry edge labeled $(_i$ from the call *Entry* vertex to the *Start* vertex of *p*, and an exit edge labeled $)_i$ from the *End* vertex of *p* to the *Exit* vertex of the call to *p*. The label $(_i$ and $)_i$ are used to maintain the calling context when computing a valid interprocedural path. In addition, an ICFG contains an edge from the *Global Start* vertex to the *Start* vertex of the main procedure—the procedure that is always executed first (start node in the call graph)—and an edge from the *Exit* vertex of the main procedure to the *Global Exit* vertex.

As for B-L paths, defining M-R paths requires some transformation on the ICFG. Figure 16 displays the transformed ICFG used to compute M-R paths. An original ICFG includes the gray edges but does not include the dashed edges. In Figure 16, the three

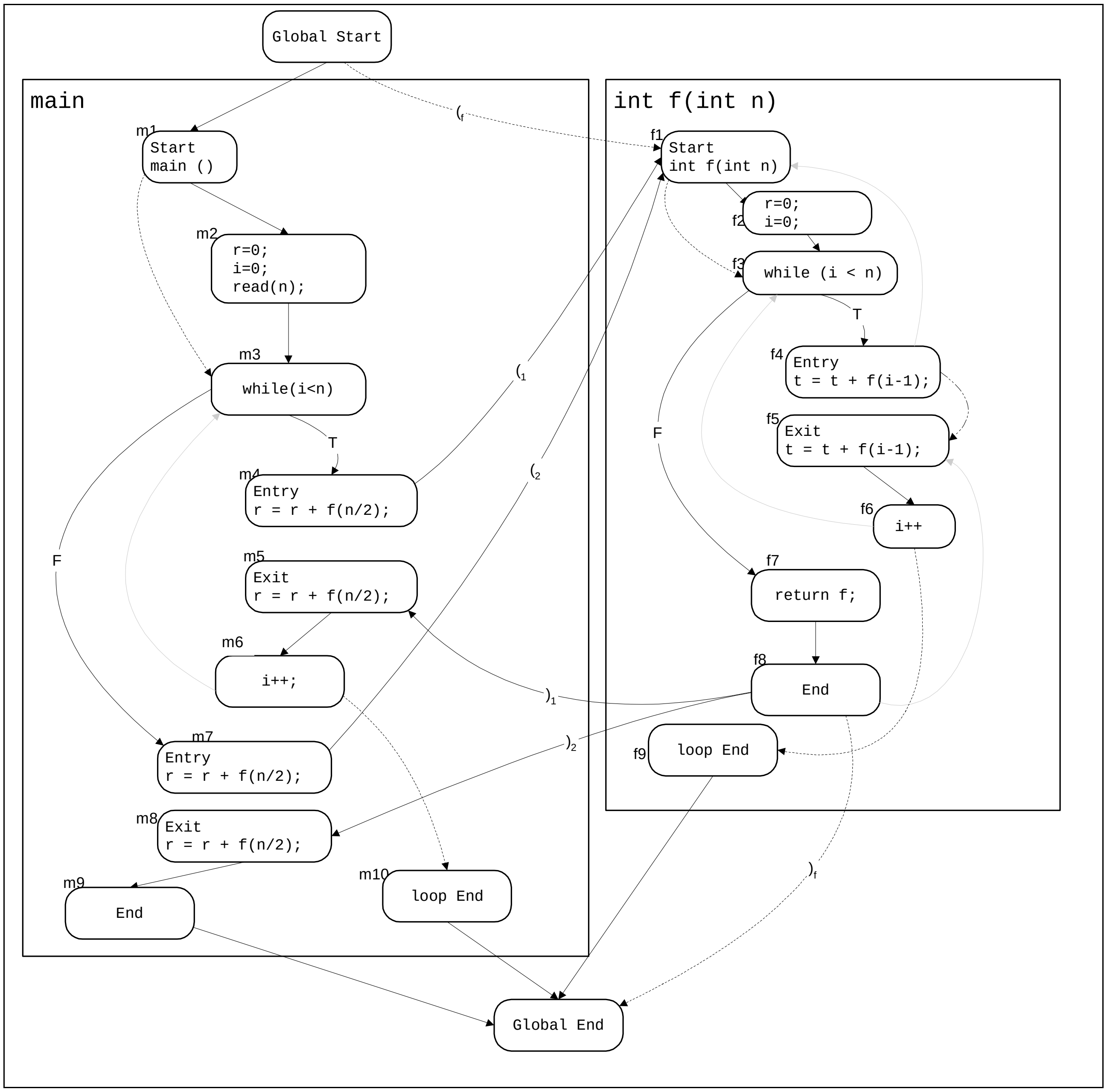


Figure 16: Interprocedural control flow graph transformed for computing M-R paths.

transformations that must be performed on an ICFG to compute the set of M-R paths are the following:

- An extra *loop End* vertex is added to each procedure. An edge from each *loop End* vertex to the *Global Exit* vertex is added.

- Every back edge (the gray edge in Figure 16) is discarded and replaced by two edges (dashed edges in Figure 16) respectively from *Start* to the source of the back edge and from the target of the back edge to *loop End*. We refer to the new dashed edges as surrogate edges. Creating these surrogate edges is the same operation required by Ball-Larus; it removes intraprocedural cycles. In terms of execution, traversing the dashed edge from *Start* vertex to the target of a back edge represents the beginning of a new iteration of a loop, and traversing the dashed edge from the source of the back edge to the *End* vertex represents the end of a loop iteration.
- Edges generated by a recursive call to procedure *p*, also shown as gray edges in Figure 16, are discarded. In the ICFG, a recursive call to a procedure *p* is responsible for two control flow edges: an edge going from the *Entry* of *p* in a procedure *m* to *Start* of *p*, and an edge going from the *End* of *p* to the *Exit* of *p* in procedure *m*. These two edges are replaced by a summary edge going from the *Entry* of *p* in *m* directly to the *Exit* of *p* in *m* (long dash edge in Figure 16). We note that this new edge is referred to as a *summary* edge, not a surrogate edge. In addition, two interprocedural edges are created: a first one from *Global Start* vertex to the *Start* vertex of procedure *p*, labeled $(_p$, and another one from the *End* vertexof *p* to *Global End*, labeled $)_p$. Edges mark $(_p$ or $)_p$ are called recursive edges.

Definition:
- The following BNF grammar helps specify the *M-R paths* in the transformed ICFG. To each M-R path, there corresponds a string of left unbalanced (or balanced) parentheses.

- Let us assign a label *e* to every edge without a parenthesis label:

  Unbalanced Left ::= Unbalanced Left $(_i$ Unbalanced Left
  Unbalanced Left ::= Unbalanced Left $(_p$ Unbalanced Left
  Unbalanced Left ::= Balanced
  Balanced ::= $(_i$ Balanced $)_i$ Balanced (for $1 \le i \le$ *number of call site in the program*)
  Balanced ::= $(_p$ Balanced $)_p$ (for a procedure *p* called recursively)
  Balanced ::= *e*
  Balanced ::= ε (where ε means empty string)

- Referring to the grammar above, we define an *M-R path* as a path from *Global Start* to *Global End* in the transformed ICFG, which corresponds to a string of balanced or left unbalanced parentheses.

In addition to the above transformations needed to define M-R paths, Melski and Reps developed an algorithm to assign a unique number to each M-R path. Moreover, they explain where and how the source code of a system must be instrumented to record the unique numbers of M-R paths exercised during a system run. Melski and Reps also show there are a finite number of M-R path in the source code of a system [Melski and Reps 1998, Melski 2002].

Table 5 lists all twenty M-R paths of the ICFG of Figure 16. We also added an invalid path in the last row of the table in order to illustrate the notion of nonmatching subscript of parentheses. For all M-R paths, Table 5 includes the parentheses and their labels to show they are valid. GS and GE respectively mean *Global Start* and *Global End*. Using the columns titled *Main*, *f*, and *Recursion on f*, we attempt to relate an M-R path to its actual execution behaviors. To explain the different execution behavior captured in an M-R path, we observe that two types of edges were introduced during the transformation on the ICFG: some intraprocedural surrogate edges and some interprocedural edges. In terms of execution behavior, traversing the former edges means repetition of a loop (intraprocedural cycle), and

Table 5: M-R paths of ICFG of Figure 16.

	M-R path explanation		M-R paths
	Main	*f*	
1	N	N	GS, m1, m2, m3, m7, $(_2$, f1, f2,f3,f7,f8, $)_2$, m8, m9, GE
2	N	1st	GS, m1, m2, m3, m7, $(_2$, f1, f2, f3, f4, f5, f6, f9, GE
3	N	O	GS, m1, m2, m3, m7, $(_2$, f1, f3, f4, f5, f6, f9, GE
4	N	L	GS, m1, m2, m3, m7, $(_2$, f1, f3,f7,f8, $)_2$, m8, m9, GE
5	1st	N	GS, m1, m2, m3, m4, $(_1$, f1, f2, f3, f7, f8, $)_1$, m5, m6, m10, GE
6	1st	1st	GS, m1, m2, m3, m4, $(_1$, f1, f2, f3, f4, f5, f6, f9, GE
7	1st	O	GS, m1, m2, m3, m4, $(_1$, f1, f3, f4, f5, f6, f9, GE
8	1st	L	GS, m1, m2, m3, m4, $(_1$, f1, f3, f7, f8, $)_1$, m5, m6, m10, GE
9	O	N	GS, m1, m2, m3, m4, $(_1$, f1, f2, f3, f7, f8, $)_1$, m5, m6, m10, GE
10	O	1st	GS, m1, m2, m3, m4, $(_1$, f1, f2, f3, f4, f5, f6, f9, GE
11	O	O	GS, m1, m2, m3, m4, $(_1$, f1, f3, f4, f5, f6, f9, GE
12	O	L	GS, m1, m2, m3, m4, $(_1$, f1, f3, f7, f8, $)_1$, m5, m6, m10, GE
13	L	N	GS, m1, m2, m3, m7, $(_2$, f1, f2,f3,f7,f8, $)_2$, m8, m9, GE
14	L	1st	GS, m1, m2, m3, m7, $(_2$, f1, f2, f3, f4, f5, f6, f9, GE
15	L	O	GS, m1, m2, m3, m7, $(_2$, f1, f3, f4, f5, f6, f9, GE
16	L	L	GS, m1, m2, m3, m7, $(_2$, f1, f3,f7,f8, $)_2$, m8, m9, GE
	R_f		
17		N	GS, $(_f$, f1, f2,f3,f7,f8, $)_f$, GE
18		1st	GS, $(_f$, f1, f2, f3, f4, f5, f6, $)_f$, GE
19		O	GS, $(_f$, f1, f3, f4, f5, f6, $)_f$, GE
20		L	GS, $(_f$, f1, f3,f7,f8, $)_f$, GE
Invalid			GS, m1, m3, m4, $(_1$, f1, f3, f7, f8, $)_2$, m8, m9, GE

traversing the latter edges means repeating a recursive procedure (interprocedural cycle.) We find that the edges introduced in the transformation of the ICFG enable the M-R path to capture the five types of execution behaviors listed below. The first four behaviors express intraprocedural behavior, and the last one expresses an inter-procedural behavior:

- A *while* loop not entered or a *do while* loop not repeated (Table 5 uses the symbol N to refer to this execution behavior),
- The first iteration of a loop (Table 5 uses 1st to refer to this execution behavior),
- Other iterations of the loop (Table 5 uses O to refer to this execution behavior),

- A loop was previously entered and it is now being exited (Table 5 uses L to refer to this execution behavior), and
- A recursion on procedure *p* (Table 5 uses R_p to refer to this execution behavior).

The last row of Table 5 is invalid because the subscripts of the parentheses do not match. In terms of execution, this would mean entering a procedure *p* from one call site and returning to another call site of procedure *p*. Obviously, this is an invalid execution. This table shows how labeled parentheses are used to maintain the calling context of a procedure call.

As previously stated, Melski and Reps proposed a method to instrument source code so that the set of M-R paths exercised during a system execution is recoded. For example, executing the program of Figure 16 with the input `‘0’` records the following set of M-R paths, where we refer to a M-R path by its row number in Table 5: `<input=0, set of M-R paths={1}>`. In this case, only one M-R path is exercised. However, for all other cases, more than one M-R path is exercised. For example, with input `‘2’` the outcome is:

```
<input=2, set of M-R paths={6,7,8,10,11,12,14,15,16, 17}>
```

3.4.3 Restriction on software functions and criterion of source code coverage

We have four different types of units of source code available, node, branch, B-L path, and M-R path. We now use these units to first define restriction Z on software functions (Section 3.4.3.1) and then describe a criterion X of source code coverage (Section 3.4.3.2).

3.4.3.1 Restriction Z on software functions

We know that in the general case we cannot guarantee safe predictions. Therefore, in this section, we analyze how to use the unit of source code to create a restriction Z on software functions. We create a definition that regroups the four different types of units of source code into one common term: *trace element*.

Definition: A ***trace element*** is a node (=source code component = basic block,) a branch, a B-L path or a M-R path.

We start with a restriction and then we weaken it until it defines a restriction Z that includes a broad category of software functions. When specifying our restriction Z, our different attempts refer to a trace element *e*. In all cases, we assume that

e is or contains the source code component *c* that in turn includes the select spot of source code where a change is proposed.

Our first restriction is the following:

First attempt: Every software function *f* must be associated to one trace element *e* such that if a system test *t* activates *f* then *t* always exercises *e* and no system test *t′* exercises *e* without activating *f*.

In other words, our first attempt requires that every software function be related to at least one trace element in a unique manner. This condition is too restrictive because *several* trace elements may be needed to express the uniqueness of source code behavior associated to a software function *f*. For example, software function *f* is applied if trace elements e_1 and e_2 are exercised. The sole exercise of e_1 without e_2 or of e_2 without e_1 would not suffice to determine whether *f* was applied or not.

To accommodate for this new possibility, our condition needs to be stated as follows.

Second attempt: Every software function *f* must be associated to a set *E* of trace elements such that if a system test *t* activates *f* then all trace elements of *E* are exercised, and no execution of system test *t′* exercises all elements of *E* without activating *f*.

Again, we must weaken the above condition. Indeed, it is possible that no trace element (or set of trace elements) is always exercised when a software function is activated. This is often true when the implementation of a software function has been completely duplicated in several areas of the source code. To allow for such a scenario, our restrictive condition must be formulated as follows.

Third attempt: For every application of a software function, there exists a set *E* of trace elements such that

> If a system test *t* exercises all trace elements of *E* then *t* activates *f*,
>
> And
>
> No execution of a system test *t′* exercises all trace elements of *E* without activating *f*.

When the implementation of software functions is not duplicated, there may exist only one set *E* of trace elements that meets the condition above. However, when such duplication exists in the source code, it is likely that several sets *E* meet our condition. Our restrictive condition currently uses the existential quantifier *there exists a set E of trace elements*; hence, it accommodates for situations where there are one or more sets *E* of trace elements that satisfies our restriction.

So far, we have ignored whether software functions are dependent or independent of each other.

Definition: Two software functions are ***dependent*** if there exists a descendant/ascendant relationship between them in a generalization/specialization hierarchy. In contrast, two software functions are ***independent*** if they are not dependent.

However, as we point out below, the same set *E* of trace elements can be associated to a software function *f* and also to all of *f*'s ancestors. Consequently, our current condition does not need further modifications.

To explain the reason why our condition does not need to be changed, we recall that the hierarchies of software functions specified in Chapter 2 are built using the generalization/specialization relationships between software functions. For instance, in our ATM example, the software function *process withdrawal from checking* is a specialization of the software function *process withdrawal*. When a system test *t* activates the software function *process withdrawal from checking*, it must also activate *process withdrawal*. In general, we know it is always the case that when two software functions *f* and *special-f* are dependent, the system test that activates *special-f* also activates *f*. In turn, this means that our condition does not need to make the distinction between *special-f* and its ancestors. In other words, a same set *E* of trace elements can be used to show that *special-f* is activated but also to show that all of *f*'s ancestors are activated.

We therefore use our third attempt to specify our restriction Z on software functions below.

Restriction Z on software functions: *For every application of a software function, there exists a set E of trace elements such that*

> *If a system test t exercises all trace elements of E then t activates f,*
>
> *And*
>
> *No execution of a system test t′ exercises all trace elements of E without activating f.*

There exists an interesting property between our restriction Z and software hierarchies. In order to express this interesting property, we refer to the notion of *complete* *specialization* of a software function. Complete specialization was defined in Section 2.1. It specifies that a set *F* of software functions is a complete specialization of a software function *f* if at least one function in *F* is activated when *f* is activated.

The interesting property is the following:

Property: IF a set *F* of software functions satisfies our restriction Z on software functions
AND
IF *F* is a complete specialization of software function *f*
THEN
f also satisfies our restriction Z on software functions

Proof:

- Let *F* be the set of software functions that is a complete specialization of *f*.
- Let E_i be the set of trace elements that show $f_i \in F$ satisfies restriction Z.
- Let *E* be the union of E_i's of every $f_i \in F$.
- Set *E* of trace elements shows that there exists a set of trace elements associated to software function *f* such that when $e \in E$ is exercised, *f* is activated and also satisfies *Z*.

3.4.3.2 Criterion X of source code coverage

We now use our four units of source code to define an adequate source code coverage criterion X. This ensures the needed connection between our criterion X and our restriction Z. Moreover, in this section, we argue that our criterion X of source code coverage is reachable by a finite set of system tests.

M-R path is the most specific of the four units of source code used to define restriction Z. Moreover, we know that a source code coverage criteria specified in terms of M-R paths subsumes coverage criteria specified by the other three units of source code—nodes, branches, and B-L paths. In Section 3.4.3.2.1, we show with generic and specific examples why our coverage criterion X must not only merely consider coverage of single M-

R paths but also consider coverage of sets of M-R paths. In turn, in Section 3.4.3.2.2, we describe our criterion X using sets of M-R paths.

3.4.3.2.1 M-R paths coverage

The first coverage that comes to mind requires every M-R paths to be exercised by at least one system test. However, as we illustrate below, this coverage criterion is not enough.

1. Let us assume that a software system has two software functions f_1 and f_2 that satisfy our restriction Z. In particular,
    - f_1 is associated to two sets of trace elements, in this case, M-R paths.

      Let us say $\{p_1, p_2\}$ and $\{p_2, p_3\}$.
    - f_2 is associated to one set of trace elements, also M-R paths.

      Let us say $\{p_4\}$.
2. Let us now assume that for the two system tests:

   t_1 exercises $\{p_1, p_2\}$ and t_2 exercises $\{p_3, p_4\}$.
3. M-R paths p_1, p_2, p_3, and p_4 have all been exercised. From our restriction Z, we know that t_1 must have activated f_1 and t_2 activated f_2.
4. However, let us assume that there exists a possible system test that exercised $\{p_2, p_3\}$, but that this test scenario was not executed.
5. In such case, if our method is used to compute a prediction for a spot in source code component *c* where *c* is only contained in M-R path p_3 then only f_2 is potentially affected by a change at the proposed spot. In fact, our method finds that only t_2 exercised p_3. Furthermore, t_2 only activated f_2. However, from point 4, we know there exists an untested scenario that would have exercised $\{p_2, p_3\}$. In turn, this would show that f_1 is also potentially affected. Therefore, our prediction is unsafe.

```
// Bank ATM
main ()
{
   card = read_card_type(card_info);
   if (card.hasChip() == False) {
      // code magnetic validation here

      another = 'y';
      while (another == 'y') {
         op_read = False;
         while (op_read == False) {
            cout << "Enter operation";
            cin >> op;
            if (op == 'W')
               op_read = True;
               op_fp = withdraw;p1
            else if (op == 'D')
               op_read = True;
               op_fp = deposit;p2
         }

         ac_read = False;
         while (ac_read == False) {
            cout << "Enter account";
            cin >> ac;
            if (ac == 'C')
               ac_read = True;
               acct = card.getChecking();p3
            else if (ac == 'S')
               ac_read = True;
               acct = card.getSavings();p4

         }

         op_fn(acct);

         cout << "Other transaction?";
         cin >> another;
      }
   }
```

```
   else { // This cards has a chip
      // code chip validation here

      another = 'y';
      while (another == 'y') {
         op_read = False;
         while (op_read == False) {
            cout << "Enter operation";
            cin >> op;
            if (op == 'W')
               op_read = True;
               op_fp = withdraw;p5
            else if (op == 'D')
               op_read = True;
               op_fp = deposit;p6
         }

         ac_read = False;
         while (ac_read == False) {
            cout << "Enter account";
            cin >> ac;
            if (ac == 'C')
               ac_read = True;
               acct=card.getChecking();p7
            else if (ac == 'S')
               ac_read = True;
               acct =
card.getSavings();p8

         }
         // Change line below
         op_fn(acct);

         cout << "Other transaction?";
         cin >> another;
      }
   }

   eject_card ();
} // end main
```

Figure 17: Sample implementation of another bank ATM.

Such a scenario is not only theoretical. Below we present an illustration where the coverage of all M-R paths is not always sufficient to compute safe predictions. Figure 17 illustrates a slight variation of the source code implementation of our bank ATM. The line indicated with ***p**$_i$*s shows the last statement of M-R paths that create the problem. In order to make this illustration more realistic, we must first explain how the source code reached its current state.

Let us assume that originally the ATM was made of the source code in the left column of Figure 17, in particular, only the source code nested in the *then* part of `‘if(card.hasChip() == False)’`. Then, bank cards with chip appeared on the market. To take advantage of the information found on the bank card chip, the source code was modified to become that displayed in Figure 17. The maintenance performed in order to produce the current code consisted of the following steps:

1. Introduce the following if-condition `‘if (card.hasChip()==False)’`.
2. Duplicate the code in the *then* and the *else* part of that if-statement.
3. Adapt the source code in the *else* part to make use of the extra information found in bank cards with chip.

We now illustrate the fact that merely considering the coverage of all the M-R paths is not enough to guarantee safe predictions.

Let us consider the four ATM software functions: *withdraw from checking*, *withdraw from savings*, *deposit in checking*, and *deposit in savings*. One could create more precise software functions that specify whether these transactions are performed with a regular magnetic bank card or with a chip bank card. However, for the user the difference in the bankcards does not affect the functionality of the ATM. In turn, we decide not to change the list of software functions.

Let us now explain the p_1, p_2, p_3, p_4, p_5, p_6, p_7, and p_8 shown in bold in Figure 17. Every p_i points to an M-R paths. Each M-R paths consists of a path from the beginning of the source code until p_i, which is the last executable statement of the M-R path p_i. After p_i, the M-R path terminates with two vertices: the corresponding *loop End* followed by *Global End*.

To cover the eight M-R paths, we only need the four system tests in Table 6 where the actual t_is are:

- t_1: withdraw $100 from savings (with a regular magnetic bank card)

Table 6: *Exercise* and *Activate* relationships sampled during the system tests t_is.

Input	Software functions applied	M-R paths exercised
t_1	*{ withdraw from savings }*	{ …, p_1, p_4 }
t_2	*{ deposit on checking }*	{ …, p_2, p_3 }
t_3	*{ withdraw from checking }*	{ …, p_5, p_8 }
t_4	*{ deposit on savings }*	{ …, p_6, p_7 }

- t_2: deposit \$100 in checking (with a regular magnetic bank card)
- t_3: withdraw \$100 from checking (with a chip bank card)
- t_4: deposit \$100 in savings (with a chip bank card)

Table 6 also shows that the four system tests activate the four software functions, *withdraw from checking*, *withdraw from savings*, *deposit in checking*, and *deposit in savings*. Thus, these four system tests cover all eight M-R paths and all four software functions.

We now show that even with such coverage, our method still computes unsafe predictions. Let us assume that a maintenance exercise proposes to modify the line of source code indicated with the comment in bold in Figure 17. Let us refer to this source code component as *c*.

Our method first identifies that t_3 and t_4 exercised *c*. In turn, t_3 and t_4 respectively activate software functions *withdraw from checking* and *deposit in savings*. Hence, our method infers that only these two software functions are potentially affected by a change in *c*. This is incorrect since the other two software functions *withdraw from savings* and *deposit in checking* can also potentially be affected. The execution of the two following system tests activates *withdraw from savings* and *deposit in checking* and also exercises source code component *c*.

- t_5: withdraw \$100 from savings (with a chip bank card)
- t_6: deposit \$100 on checking (with a chip bank card)

In conclusion, our generic and specific illustrations show that measuring single M-R path coverage is not enough to guarantee safe predictions. In the next section, we define another more thorough source code coverage to remedy this problem.

3.4.3.2.2 Coverage of sets of M-R paths

In order to define this new type of coverage, we first observe that a system test exercises not a single M-R path but a set of M-R paths. Hence, our new coverage is built using sets of M-R paths.

Definition:

- A ***complete path*** in the (untransformed) ICFG is a finite sequence of control flow edges that starts by the edge from *Global Start* and terminates by the edge to *Global End,* and there is a corresponding string of balanced parentheses.
- A ***combination of M-R paths*** is ***complete*** if the union of all the M-R paths in the combination corresponds to a complete path in the (untransformed) ICFG.

Lemma: *The set of all valid combinations of M-R path is finite.*

Proof: There are a finite number of M-R paths. Thus, the set of all combinations of M-R paths (or the power set of M-R paths) is also finite. The set of all complete combinations of M-R paths is a subset of the power set of M-R paths; therefore, it is finite.

In this work, we limit our effort to identifying a set of properties needed for guaranteeing that our method computes safe predictions. We leave for future work the computation that extracts all the complete combinations of M-R paths from the power set of M-R paths. In fact, before developing such a method, we have another problem to solve.

Even for small systems, the number of M-R paths is huge. The number of complete combinations of M-R paths is even larger. It is therefore totally unpractical to require a set of system tests to exercise all complete combinations of M-R paths. One way to solve this problem is by executing a limited number of system tests to obtain a few *Potentially Affect* relationships between software functions and source code. We refer to these few relationships as seeds. Using these seeds, heuristics would then infer new correspondences between these software functions and the unexercised M-R paths. Developing the needed heuristics is also left for the future. Currently, we direct our attention to identifying a property that a set of system tests must have for the seeds to be reliable. Indeed, in order for heuristics to infer new correct correspondences, seeds must be reliable. The work presented in Chapter 4 is our attempt at providing a set of criteria for selecting a set of system tests that will then be exercised to collect seeds of value.

Before giving our criterion X of source code coverage, we observe that the conditions of different branches and different loops are sometimes related. Due to these dependences, certain complete paths in the (untransformed) ICFG can never be executed. We say that such complete paths are unrealizable. It is possible for a complete combination of M-R paths to be associated only to unrealizable complete paths. Statically computing the unrealizable paths is unsolvable, for some important information may depend on the value of inputs. As mentioned in the paragraph above, in the future, we intend to develop heuristics for inferring new correspondences for a few seeds. These heuristics will solve the problem of unrealizable paths. Indeed, these heuristics will be able to associate software functions to a set of M-R paths corresponding to unrealizable paths if needed. For the moment, we assume that combinations of M-R paths that only correspond to unrealizable paths are removed from the set of complete combinations. This ensures that every complete combination of M-R paths has a corresponding realizable path. In turn, we know that for every complete combination of M-R paths, there must exist a system test that exercises that particular complete combination.

Therefore, our criterion X of source code coverage requires the following:

Criterion X of source code coverage	Every complete combination of M-R paths must be exercised by the execution of at least one system test.

3.4.4 Reaching our new goal

Showing that restriction Z and criterion X allow reaching our goal requires a proof that

1. There exists a finite set of system tests that satisfies criterion X of source code coverage and criterion Y of software function coverage for all software functions that respect restriction Z. This part is shown in Section 3.4.4.1.
2. When a set T of system tests satisfies criterion X and Y, the *Exercise* and *Activate* relationships sampled using *T* guarantee that our method computes safe predictions for the category of software functions that respect restriction Z. This part is shown in Section 3.4.4.2.

Let us first recall criteria X, Y and restriction Z.

Criterion X of source code coverage	*Every complete combination of M-R paths must be exercised by at least one system test.*
Criterion Y of software function coverage	*Every software function must be activated by at least one system test.*

Restriction Z on software functions *For every software function f, every time f is activated there exists a set E of trace elements such that*

> *If a system test t exercises all trace elements of E then t activates f,*
>
> *And*
>
> *No execution of a system test t' exercises all trace elements of E without activating f.*

3.4.4.1 Satisfying the first point of our new goal

To show that there exists a finite set of system tests that satisfies criterion X of source code coverage and criterion Y of software function coverage, we proceed as follows. First, we show that there exists a finite set *T* of system tests whose execution satisfy criterion X. Second, we show that for all software functions that respect restriction Z, satisfying criterion X implies satisfying criterion Y. Hence, the finite set *T* of system tests that satisfied criterion X also satisfies criterion Y.

Lemma: *There exists a finite set of system tests whose execution satisfies criterion X of source code coverage.*

Proof: Criterion *X* specifies that

> *Every complete combination of M-R paths must be exercised by at least one system test.*

1. We know that there exists a finite number of *valid combinations of M-R paths.*
2. By definition, we also know that for every *valid combination C of M-R paths,* there exists a system test whose execution exercise *C.*
3. From 1 and 2, there exists a finite set of system tests that exercise all complete combinations of M-R paths.

Lemma: *For all software functions that respect Z, the set T of system tests that satisfies criterion X also satisfies criterion Y.*

Proof: Criterion Y specifies that

Every software function must be activated by at least one system test.

1. By contradiction, let us assume there exists a software function *f* that was not activated by *T*.
2. From restriction Z, we know that every software function *f* is associated to at least one set *E* of trace elements. Moreover, every *e* in *E* is a node (source code components), a branch (pair of source code components), a B-L path (an acyclic intra-procedural sequence of source code components), or a M-R path (an inter-procedural sequence of source code components). We also know that for every node, branch, and B-L path there exists an M-R path that contains it.
3. Let $E_1, \ldots, E_n$ be the sets of trace element associated with *f*.
4. CASE 1: For a set E_i in $E_1, .., E_n$, there exists a *valid combination C of M-R paths* that contains all trace elements of E_i. In such a case, from our restriction Z and criterion X, we know *f* must have been activated. This contradicts our assumption in point 1.
5. CASE 2: For every set E_i, all trace elements of E_i are never found in a *valid combination C of M-R paths*.
    1. In such a case, there is no realizable path for which software function f is activated. In other words, every set E_i's describes an unrealizable path.
    2. Hence, software function f can actually never be performed by the system.
    3. By definition of software function, software function f must be a task performed by the system.
    4. From point above, f is not a software function. This contradicts our assumption in point 1 that states that f is a software function.
6. Both cases above contradict point 1; hence, the set *T* of system tests must satisfy criterion Y.

From the two lemmas above, we can infer that the first point of our goal is satisfied. In other words, there always exists a finite set *T* of system tests that satisfies criterion X of source code coverage, and criterion Y of software function coverage for all software functions that respect restriction Z.

3.4.4.2 Satisfying the second point of our new goal

We now must show that:

IF a set *T* of system tests satisfies criterion X of source code coverage

and criterion Y of source code coverage

THEN when *Exercise* and *Activate* relationships are sampled using T, our method computes safe predictions for all software functions that respect restriction Z.

In our proof, we assume that the set *T* of system tests that satisfies criteria X and Y is provided and that the *Exercise* and *Activate* relationships resulting from the execution of *T* have also been sampled. Thus, our attention focuses on the latter part of the statement above. That is, we want to show that *our method computes safe predictions for all software functions that respect restriction Z.*

Instead of showing that every prediction is safe, we use a proof by contradiction. In other words, we suppose that our method computes an unsafe prediction for a particular spot in a source code component *c*. Then, we show that our supposition cannot be true.

In order for our method to compute an unsafe prediction, the following scenario must take place. Our method computes a set *A* of software functions as being potentially affected by a change to source code component *c*. However, a software function *f* that respects restriction Z is also potentially affected by a change to *c*, and *f* is not in set *A*.

Assumptions:	1. The execution of set *T* of system tests satisfies criterion X, that is, it all complete combinations of M-R paths are exercised. 2. The *implementation participation* assumption is true. That is, if a source code component *c* ***participates in the implementation of*** software function *f* then there must exist a system test *t* that exercises *c* and activates *f*. 3. Set *T* is used to sample *Exercise* and *Apply* relationships, and then, *Potentially Affect* relationships are inferred by joining the *Exercise* and *Apply* relationships sampled.
Theorem:	*When a set T of system tests satisfies criterion X and Y, then the Exercise and Activate relationships sampled using T guarantees that our method computes safe predictions for the category of software functions that respect restriction Z.*
Proof:	We proceed by contradiction. 1. Let set *A* of software functions be a prediction computed using the *Potentially Affect* relationships for a source code component *c*. Moreover, let us assume that *A* does not contain a software function *f* and *f* is potentially affected by a change in source code component *c*. 2. If *f* is potentially affected by a change in *c* then source code component *c* must participates in the implementation of *f*. 3. From assumption 2, there must exist a system test *i* that activates software function *f* and exercises source code component *c*. 4. From restriction Z, we know that if *i* activates *f*, there must exist an associated set E_f of trace elements such that $\forall e \in E_f$ are exercised when *i* is executed. 5. From 3 and 4 above, *i* exercises all the trace elements $e_f s \in E_f$ and exercises *c*. Hence, there must exist a realizable complete path that contains every trace element in E_f and also contains *c*. 6. Hence, there exists a complete combination *C* of M-R paths where each trace element in E_f is found in at least one M-R path of *C* and where *c* is also found in at least one M-R path of *C*. 7. From assumption 1, our set *T* of system tests must contain a system test *j* that exercised *C*. *j* may or may not be equal to *i*. In any case, *j* exercises *C*, which exercised every trace element in E_f ; hence, *j* must activate *f*. 8. From 7, based on a system test *j*, our method must have included *f* in its prediction. 9. Point 8 contradicts point 1; hence, under the stated assumptions, our method must compute safe predictions for all software functions that respect restriction Z.

3.5 Assessment of our solution

In Chapter 2, we point out that the quality of a solution is determined by safety, precision, and practicality. In particular, does the solution allow our method to compute safe predictions? Precise predictions? Are the criteria required by the solution practical?

Concerning the first factor of safety of predictions, we find that our solution is satisfying. In fact, we have found a set of conditions under which our method computes safe predictions for a well-defined, broad set of software functions.

Our restriction on software functions seems adequate for our solution to be considered practical. That is, many software functions of many software systems naturally respect the restriction. On the other hand, a further analysis of the coverage conditions required by our solution shows that it is currently not practical. In particular, the source code coverage criterion requires that the system tests exercise all complete combinations of M-R paths. Although the number of complete combinations of M-R paths is finite, their number is large, even for small systems. It is therefore unrealistic to require the exercise of all these possibilities.

Although currently impractical, our solution provides a finite bound to the problem of computing *Potentially Affected* relationships. As mentioned earlier in this chapter, we plan on developing a technique where a limited number of system tests are first executed to collect some *Potentially Affect* relationships (seeds) between software functions and source code. Then, some heuristics will use the seeds to infer new *Potentially Affect* relationships between software functions and unexercised complete combination of M-R paths. For heuristics to compute reliable relationships, the seeds must provide reliable information. In other words, before developing the heuristics needed for inferring new *Potentially Affect* relationships, we must first find a technique to obtain safe and precise seeds. The next chapter works in that direction. It specifies a set of criteria for selecting a few system tests that will hopefully provide safe and precise *Potentially Affect* relationships between software functions and source code.

The last qualitative factor of a solution is whether or not it enables finding precise predictions. The solution developed in this chapter neither focuses nor mentions precision of predictions. Its objective was only geared toward computing safe predictions. Nevertheless, if all valid combinations of M-R paths are related to software functions, we know that much of the precision of predictions will be lost. Hence, the heuristics mentioned above will not only have to infer new *Potentially Affect* relationships, but they will also have to be tailored to maintain an acceptable level of precision for predictions. In particular, instead of inferring new *Potentially Affect* relationships for all complete combinations of M-R paths, the heuristics will need to eliminate the complete combinations of M-R paths whose exercise decrease precision while not adding to safety of predictions.

In conclusion, our solution shows that our method computes *safe* predictions for a well-defined, broad set of software functions using only coverage information. On the positive side, the coverage needed for computing safe predictions is reachable by a *finite* set of system tests. However, that set of system tests is of unpractical size. Moreover, our current solution makes no guarantee as to the precision of predictions.

In the next chapter, we propose a new solution that remedies the disadvantages of our current solution. In particular, our new solution wants to improve on the precision of predictions and makes sure that these results can be obtained using a small number of system tests. We concede that our new solution cannot guarantee safe prediction; however, safety remains acceptable. This new solution can be used directly by our method for computing *Potentially Affect* relationships. However, the original indent is for this new solution to provide a few reliable *Potentially Affect* relationships (seeds). Then, these seeds can be used by heuristics for inferring new *Potentially Affect* relationships of high reliability for the entire source code, even the one not covered by the seeds.

4 A practical application of our method

In this chapter, we define criteria for selecting a few system tests that provide reliable *Potentially Affect* relationships. We know that with just a few system tests, the safety and the precision of predictions are not guaranteed. However, we want the information captured in the sampled *Potentially Affect* relationships to be as safe and as precise as possible. In any case, applying our method this way is practical since it only requires a few system tests. The information obtained by these few system tests may be used directly. However, the original intent is for these system tests to provide seed information that heuristics will be able to use to infer new reliable *Potentially Affect* relationships.

This chapter is organized as follows. First, we present a set of criteria for selecting system tests. Then, we perform a case study to evaluate the criteria in the context of our method.

4.1 *Criteria for system test selection*

When identifying the criteria needed for system test selection, the guidelines are the following:

1. Few system tests must be needed to satisfy these criteria.
2. We prefer qualitative over quantitative coverage. By quality, we mean coverage that allows our method to compute safe and precise predictions.

The second point applies specifically to source code coverage. For software function coverage on the other hand, we know that every software functions must be activated at least once. In fact, given the way our method computes predictions, we know that nothing can be predicted about nonactivated software functions. In contrast, for source code coverage, we know that a few system tests can only exercise a limited amount of paths in the source code. Instead of trying to spread out the source code coverage achieved by these system tests, like software testing often requires, we prefer that the source code components exercised by the system test stay concentrated. This would usually guarantee safer and precise prediction by our method. Therefore, instead of trying to maximize coverage of source code components, we prefer that the exercise of source code remains focused.

Consequently, we specify our criteria for system test selection as follows.

1. Every software function of the system must be activated during at least one system test. When using Sonar to compute predictions, this criterion translates to the following: each software function found in Sonar's *software function specification* file must be activated by at least one system test.
2. When possible and appropriate, different system tests must reuse the same data values.
3. System tests must avoid composition of software functions as much as possible; i.e., they should only activate one software function when possible. However, we know that some software functions require the prior activation of other software functions. In such case, software function composition is acceptable and required in order to satisfy criterion 1.
4. Criteria 1 above must be satisfied with the least amount of system tests possible.

Criteria 1 and 2 increase the size of the set of system tests that will be used for sampling *Exercise* and *Activate* relationships. In contrast, criteria 3 and 4 restrict the number of system tests that will be selected. Using such guidelines, we make the set of system tests a controlled dependent variable of our studies. We refer to the set of four criteria above as the *test selection criteria*.

4.2 Assessing our test selection criteria through case study

We now assess the level of prediction safety and correctness when computed by our method with the *Exercise* and *Activate* relationships sampled from a set of system tests that satisfy our test selection criteria. A prediction is correct or exact if it is 100% safe and 100% precise. This study uses Sonar to compute the predictions. We repeated the study on two software systems, in particular *scalc* and *bool*.

We present our case study as follows. In Section 4.2.1, we state our objective. In Section 4.2.2, we explain the protocol followed for the study. In Section 4.2.3, we present the two software systems, *scalc* and *bool*. Moreover, we enumerate a broad list of software functions for both systems. In Section 4.2.4, we present our results, and in Section 4.2.5, we draw conclusion of our studies.

4.2.1 Objectives of the study

In our study, we measure whether or not predictions are safe and if they are exact. Therefore, concerning safety, we find that a prediction is safe or unsafe. Similarly, we find that a precision is either correct or incorrect. In other words, in our study, to be considered safe and exact, a prediction must be 100% safe and 100% exact respectively.

When conducting the case study, it is unpractical to verify whether all of Sonar's predictions are safe and exact. In fact, even for small systems the number of source code components exercised is in the hundreds. Hence, we compute predictions for a pool of twenty-five randomly selected source code components, and then we infer the results for the rest of the exercised source code components. The fact that we do not compute predictions for all exercised source code components introduces some level of uncertainty in our claims. We want this uncertainty factor to remain very low, 1% or less. In turn, we now state our goal for the case study of scalc and of bool as follows.

Goal: We want to determine with more that 99% certainty that for software system Z, Sonar computes X% of safe predictions and Y% of exact predictions when *Exercise* and *Activate* relationships are sampled with a set of system tests that satisfy our test selection criteria.

Assumption: We assume that our study does not contain any error of the following type: A result computed by Sonar is said to be unsafe (or unprecise) when it is actually safe (or precise.) Thus, we are assuming a 0% β error factor.

Dependent variables

The dependent variable is the set of system tests used to sample *Exercise* and *Activate* relationships. However, thanks to our test selection criteria, we specify some control on this dependent variable.

4.2.2 Protocol used in the study

Here are the steps followed during the study:

1. For the software system selected, we do the following:
   a. Compile the selected software system in order to produce an instrumented executable version.
   b. Create a list of software functions for the software system selected.
   c. Identify a set of system tests that satisfies our test selection criteria and then execute each system test with the instrumented executable.

2. Collect the set of all source code components covered then randomly select twenty-five of these source code components. Finally, we identify the file name and the beginning line-column position of each of the twenty-five source code components.
3. Perform an initial manual analysis for each of the twenty-five positions. In particular, for a position p, we refer to the list of software functions and mark each software function we believe is potentially affected by a change at p. For this initial analysis, we do not run the system. We only read the source code and use the grep command to navigate in source code files.
4. Let Sonar compute its prediction for each of the twenty-five positions.
5. Compare the manual predictions with those computed by Sonar. For a particular position, if for a given source code position a manual prediction and Sonar's prediction are the same, we assume that they are both correct (that is, safe and precise). If two predictions are different then we perform the next two steps of the protocol.
6. Perform a second more thorough manual analysis. During this second manual analysis, we are permitted to execute the system, instrumented manually with print statements in order to determine an execution's dynamic behaviors. If needed, we adjust the first manual predictions. At this point, we believe that all of the manual predictions are exact.
7. Given the second manual exact prediction, we now determine whether or not Sonar's predictions are safe and then whether or not they are exact.

These last two steps do not need to be performed when the first manual predictions agree with Sonar's predictions. In fact, in such cases, we assume that Sonar's prediction was safe and precise; hence, it is correct. On the other hand, for the other cases where step 5 finds that two predictions are different then, we now compare Sonar's prediction to our second manual prediction. After all these steps, we are able to determine whether or not Sonar's predictions are safe and whether or not they are exact for each of the twenty-five source code components.

One may argue that our second manual analysis is influenced by Sonar's prediction since we already compared the prediction of a first manual analysis with that of Sonar's. However, this influence is irrelevant. In this case, the important factor is that we do not change Sonar's predictions. In fact, in step 7, we compare the new manual predictions to Sonar's predictions created in step 4. In other words, during steps 6 and 7, manual predictions may be changed in order to correct them, but Sonar's prediction may not be changed by sampling additional *Exercise* and *Activate* relationships.

The only valid argument on the validity of our study is that our second manual analysis may still be incorrect and that further adjustment of the manual predictions must be performed. This reasoning might be true for large systems. However, as we will see in the next section, the two software systems selected for the study have source code of small size (between 2,000 and 5,000 lines of source code including comments). We therefore believe that the assumption that *"predictions obtained by our second manual analysis are correct"* is fair for small size source code.

4.2.3 The two software systems studied: *scalc* and *bool*

4.2.3.1 scalc

scalc is an interactive spreadsheet program that uses the *curses* library to allow the user to move around the spreadsheet with the arrow keys. *scalc* is written in C++ and is based on a well-crafted object-oriented design that separates the GUIs from the core computation of the spreadsheet. Furthermore, the computation is separated between the calculator engine and the spreadsheet document. The source code size is about two thousand lines including comments.

scalc provides the software functions to perform the following tasks: (1) move around the spreadsheet, (2) load an existing spreadsheet, (3) save a spreadsheet, (4) clear a spreadsheet, (5) recalculate (or reevaluate) a spreadsheet, (6) toggle the auto reevaluation of a spreadsheet between on and off, and finally (7) edit cell of the spreadsheet.

scalc distribution comes with the file `README.txt`, which I used for creating the list of software functions presented in Table 7. The left column of the table lists the names of the software functions and presents them in a tree-like format. For example, the software function *right* appears as a child of *motion* meaning that *right* is a particular type of *motion*. The right column provides a short description of each software function.

Table 7: List of software functions of the *scalc* software.

	Software functions	Description of the software functions
	SCALC	
1	+ Motion	Function that refers to motion from cell to cell in the spreadsheet.
2	+ Right	Motion associated to the *right* arrow key.
3	+ Left	Motion associated to the *left* arrow key.
4	+ Down	Motion associated to the *down* arrow key.
5	+ Up	Motion associated to the *up* arrow key.
6	+ Load	Function that refers to the

		loading of an existing spreadsheet.
7	+ Load Process	Once the user has selected the function load spreadsheet, the user decides to proceed with loading an existing spreadsheet.
8	+ Load Cancel	Cancel loading of spreadsheet and return to current spreadsheet.
9	+ Save	Function that refers to the saving of the current spreadsheet.
10	+ Save Process	Once the user has selected the function *save spreadsheet*, the user decides to proceed with saving the current spreadsheet.
11	+ Save Cancel	Cancel saving of the current spreadsheet and return to it.
12	+ Clear	Clear all cells of the current spreadsheet.
13	+ Process	Proceed with clearing the current spreadsheet.
14	+ Cancel	Cancel the clearing operation.
15	+ Recalculate	Recalculate the content of each cell of the spreadsheet.
16	+ Toggle Auto calc	Toggle the auto recalculate of the spreadsheet to ON or OFF. When this switch is ON, the spreadsheet updates all the required values after the edition of a cell. It only calculates the value of the current cell if the switch is OFF.
17	+ Cell edit	Enter in the cell edition mode.
18	+ Cancel edition	Cancel any edition to this cell, restore its old value, and return to current spreadsheet.
19	+ Process Edition	Update the value of the cell to the newly edited value.
20	+ Text	The format of the new value is TEXT.
21	+ Math. expression	The format of the new value is a mathematical expression that requires evaluation.
22	+ Number	The expression contains a number
23	+ Arith. Exp	The expression is an arithmetic expression (contains +, -, /, or * operators).

24	+ Function	The expression contains a function such as sin, cos, tan, atan, sqrt, etc.
25	+ Cell ref.	The expression contains a cell reference.

4.2.3.2 bool

bool Version 0.1.1 can be downloaded from the GNU Software Foundation website (http://www.gnu.org/directory/Bool.html). *bool* is command-line driven and allows the user to search for a Boolean-expression pattern in a list of files. *bool* is written in C, and its source code size is about five thousand lines including comments. The implementation is procedural in nature. The *man* page, which was used to create the list of software functions in Table 8, specifies that *bool* takes three types of parameters:

1. Flags (or options) allow the user to activate different software functions such as ignore case, count number of matches, etc.
2. A pattern in the form of a Boolean expression made of character strings grouped using Boolean operators AND, OR, plus NEAR (default 10 words a part).
3. A series of files that are matched against the Boolean pattern. Files may be in text or html format.

These three parameters may be used when creating a list of software functions for *bool*. Furthermore, the *man* page explains that in addition to performing regular matching of patterns, *bool* also performs special matching when a pattern is split on different lines. In particular, for a text file, when a pattern starts at the end of one line and terminates at the beginning of the next line, *bool* finds a match. However, if there are several new lines that split the pattern in a text file then *bool* considers there to be no match. For html files, the rules are different. The determinant factors are html tags. For example, a pattern that is split by text formatting tags such as bold <*B*> and new line will still be considered matches;

however, when the pattern is found in the file but split by tags such as new paragraph (*<P>*) or new heading, there is no match. For example, for the pattern *'Pattern'*, *bool* finds a match for the html excerpt *<B>P<\B>attern*; however, there is no match with *<P>Pat</P>tern*.

Given that the *man* pages explain these different type of matching, one may define software functions in relation to *regular matching* and *special matching (pattern found on several lines)*. These software functions are special because they embed the notion of success. In other words, the software functions *find a pattern with a regular match* or *find a pattern with a special match*, which means that the pattern must be found in the input files. So, unlike software functions based on the three types of input parameters accepted by *bool*, these last types of software functions are not directly visible from the command line. This explains why we have found that the manual analyses involving regular and special matches were much harder than for the other types of software functions. In addition, since both regular and special match imply success, we also list the software function *fail search*. However, we find that it does not make sense to differentiate between a fail search of regular pattern and a fail search of special pattern (a failure is a failure regardless).We briefly explain how software functions can be specialized with an example, and then we present the software functions in Table 8. When creating a list of software functions for bool, one may be more or less precise by specifying a software function in terms of one, two, or all three of the types of parameters plus whether it is a regular or a special match. For example, going from general to specific (1) the software function *find regular match,* (2) *find regular match in an html file,* (3) *find regular match in an html file with a case insensitive search,* and (4) *find regular match in an html file with a case insensitive search for an ANDed Boolean expression*.

Table 8: List of software functions of the *bool* software.

	Software functions	Description of the software

		functions
	Bool	
1	+ One-word pattern search	Successful search file(s) for a one-word pattern.
2	+ Text file	Successful search text file(s) for a one-word pattern.
3	+ Regular file	Successful search text file(s) for a one-word pattern and the file(s) do not contain split patterns.
4	+ Count	Successful search text file(s) for a one-word pattern and print the numbers of matches.
5	+ Ignore case	Successful case-insensitive search of text file(s) for a one-word pattern.
6	+ N matches	Successful search text file(s) for N first occurrence of a one-word pattern.
7	+ Fixed string	Successful search text file(s) for a fixed one-word pattern (ignore the particular meaning of and, or, near).
8	+ Special file	Successful search of text file(s) for a one-word pattern where the file contains the particular pattern split on two lines.
9	+ Fail	Unsuccessful search of text file(s) for a one-word pattern.
10	+ Html file	Successful search html file(s) for a one-word pattern.
11	+ Regular File	Successful search html file(s) for a one-word pattern and the files do not contain split patterns.
12	+ Count	Successful search html file(s) for a one-word pattern and print the numbers of matches.
13	+ Ignore case	Successful case-insensitive search of html file(s) for a one-word pattern.
14	+ N matches	Successful search html file(s) for N first

		occurrence of a one-word pattern.
15	+ Fixed string	Successful search html file(s) for a fixed one-word pattern.
16	+ Special file	Successful search html file(s) for a one-word pattern where the file contains the particular pattern split on two lines.
17	+ Fail	Unsuccessful search of html file(s) for a one-word pattern.
18	+ AND’ed Search	Successful search for an AND’ed pattern.
19	+ Text file	Succssful search a text file for an AND’ed pattern.
20	+ Html file	Successful search a html file for a AND’ed pattern.
21	+ OR’ed Search	Successful search for a OR’ed pattern.
22	+ Text file	Successful search a text file for an OR’ed pattern.
23	+ Html file	Successful search a html file for a OR’ed pattern.
24	+ NEAR’ed Search	Successful search for a pattern that contains a NEARed expression.
25	+ Text file	Successful search a text file for a NEAR’ed pattern.
26	+ Html file	Successful search a html file for a NEAR’ed pattern.

4.2.4 Result of the study on *scalc* and on *bool*

After our set of system tests, nineteen system tests for *scalc* and twenty-one system tests for *bool*, we found that 419 and 508 source-code components were exercised for *scalc* and for *bool*, respectively. We then randomly selected twenty-five source-code components for each of the two systems. Finally, we identified the file name and the beginning line-column position of each of these twenty-five randomly selected source-code components. Table 9 presents the two lists of twenty-five spots of source code.

Table 9: List of spots in the source code used for our case study.

	scalc	bool
1	Calculator.cpp, 185, 5	Kw.c, 545, 17
2	Document.cpp, 271, 21	Ac.c, 348, 7
3	Document.cpp, 231, 5	Ac.c, 205, 11
4	Textview.cpp, 114, 9	Ac.c, 318, 3
5	Document.cpp, 272, 2	Kw.c, 657, 23
6	Document.cpp, 286, 5	Kw.c, 440, 5
7	Calculator.cpp, 235, 26	Ac.c, 327, 11
8	Textview.cpp, 83, 25	Kw.c, 619, 54
9	Textview.cpp, 87, 18	Sgml.c, 554, 3
10	Calculator.cpp, 307, 3	Html.c, 531, 3
11	Textview.h, 59, 23	Kw.c, 574, 19
12	Textview.cpp, 60, 1	Html.c, 404, 11
13	Textview.cpp, 492, 5	Kw.c, 334, 3
14	Textview.cpp, 632, 5	Sgml.c, 180, 7
15	Textview.cpp, 243, 5	Kw.c, 178, 7
16	Calculator.cpp, 172, 1	Mem.c, 127, 1
17	Textview.cpp, 84, 25	Sgml.c, 555, 27
18	Textview.cpp, 437, 9	Kw.c, 452, 7
19	Calculator.cpp, 454, 5	Kw.c, 680, 23
20	Textview.cpp, 628, 5	Kw.c, 421, 11
21	Calculator.cpp, 558, 5	Text.c, 124, 11
22	Document.cpp, 84, 25	Ac.c, 407, 3
23	Textview.cpp, 220, 1	Html.c, 453, 3
24	Textview.cpp, 75, 5	Ac.c, 122, 1
25	Calculator.cpp, 364, 16	Ac.c, 368, 1

4.2.4.1 Results for scalc

After performing a first manual analysis, we run Sonar. Table 10 presents the results of our first manual analysis as well as those computed by Sonar. Each row starts with a number that cross-references to the particular source code position of interest listed in Table 9. An empty cell in the Sonar column means that Sonar's prediction is the same as the manual prediction. The rows in bold indicate a discrepancy in the two predictions.

Table 10: Comparison of the results of a manual analysis and of Sonar for *scalc*.

Pos.	Manual Results	Sonar's results that help correct manual results
1	**Whole cell editing sub tree + Recalculate**	**Whole cell editing sub tree**

2	Load processed	
3	Save processed	
4	**Visual display affected**	**All software functions**
5	Load processed	
6	Load processed	
7	**Edit cell reference + Edit function**	**Edit cell reference + Edit function + Edit cell with text**
8	Motion up	
9	Whole Cell editing subtree	
10	**Edit number**	**Edit number + edit arith. exp**
11	Visual display affected	
12	Visual display affected	
13	**Toggle auto-recalculate**	**All software functions**
14	**Whole load subtree**	**Only Load processed affected**
15	**All visual display affected**	**All software functions**
16	Load processed + Recalculate	
17	Motion down	
18	**Visual display affected**	**Only Visual display after a cell edition**
19	**Whole edit expression processed sub-tree**	**Whole edit expression processed sub-tree except edition of cell reference**
20	Load processed	
21	Whole edit expression processed subtree	
22	**Clear sheet processed**	**Clear sheet processed+ Load processed**
23	Motion right	
24	**Visual display affected**	**All software functions**
25	**Whole edit expression processed subtree**	**Whole edit expression processed subtree except edition of cell reference**

Table 10 shows that there are twelve discrepancies and thirteen predictions that are the same between the manual analysis and Sonar's. As indicated in our protocol, we assume that the thirteen same predictions are exact. However, for the inconsistent predictions, we now perform a second manual analysis to determine whether Sonar's predictions are unsafe or safe and exact or inexact. Conversely, this also helps determine when the first manual analysis is unsafe or safe and exact or inexact. In Table 11, rather than only showing the case of discrepancies, we present Table 10 with all the correct results, and then we indicate whether Sonar's predictions are CORRECT, UNSAFE, or SAFE. For safe and unsafe results, we also give the extra and missing software functions, respectively. Moreover, to indicate a change in a manual result from the first to the second analysis, we italicized the results in Table 11. In other words, this points out when our first manual analysis was wrong. It happened in four cases, respectively in rows 7, 14, 18, and 22.

Table 11: Estimation of Sonar's results for *scalc*.

Pos.	Correct Results from second analysis	Sonar's results
1	Whole cell editing sub tree + Recalculate	**UNSAFE**: Missing recalculate
2	Load processed	Correct
3	Save processed	Correct
4	Visual display affected	*Correct:* indicates all affected
5	Load processed	Correct
6	Load processed	Correct
7	*Edit cell reference + Edit function + Edit cell with text*	Correct
8	Motion up	Correct
9	Whole Cell editing sub-tree	Correct
10	Edit number	SAFE: indicates edit arith. Exp affected
11	Visual display affected	Correct (none affected)
12	Visual display affected	Correct (none affected)
13	Toggle auto-recalculate	SAFE: indicates all affected
14	*Load processed*	Correct
15	All visual display affected	*Correct:* indicates all affected
16	Load processed + Recalculate	Correct
17	Motion down	Correct
18	*Visual display of cell edition affected*	*Correct:* indicates all edit cell sub-tree
19	Whole edit expression processed sub-tree	**UNSAFE:** Missing edit cell reference
20	Load processed	Correct
21	Whole edit expression processed sub-tree	Correct
22	*Clear sheet processed + Load processed*	Correct
23	Motion right	Correct
24	Visual display affected	*Correct:* indicates all affected
25	Whole edit expression processed sub-tree	**UNSAFE:** Missing edit cell reference

First, we note that three of Sonar's predictions are unsafe (1, 19, and 25). Second, two predictions (10 and 13) are safe but not exact. Finally, the remaining twenty predictions made by Sonar are correct. For results (4, 15, 18, and 24), we find that the results are correct; however, some additional interpretation is needed. The correct results indicate that only the visual display of the application is affected. Currently, Sonar cannot make the difference between the visual and the computational aspect of a software function. Hence, Sonar highlights a software function independent of whether its computational or visual aspect is affected. Programmers do not usually have problems determining if the source code they are analyzing deals with the computational part or the user interface (UI) part of a system. The tough part of the programmer’s job is to determine the particular software functions to which a particular source code component relates. Consequently, in these four cases, we find that Sonar’s predictions are safe and correct.

The results above only provide information for 25 of the 419 source-code components exercised. However, we can use the binomial distribution to estimate what the predictions would be for the remaining 396 source-code components. The binomial distribution can be used when a trial, in this case a prediction computed for a source code component, is independent of the others. This is also known as Bernoulli trials. In our case, predictions are independent of the proximity between source code components. Two source code components may generate different predictions whether they are near each other or not. Hence, predictions for different source code components are Bernoulli trials.

Finally, using the binomial distribution, we can determine the X and Y of our objective.

Goal: We want to determine with more that 99% certainty that for software system Z, Sonar computes X% of safe predictions and Y% of exact predictions when

Exercise and *Activate* relationships are sampled with a set of system test that satisfy our test selection criteria.

We may say that for the *scalc* software and our 19 system tests that satisfy our test selection criteria

- X = 70%. We know with more than 99% certainty that 70% of Sonar's predictions for the remaining 396 source code components would be safe
- Y = 60%. We know with more than 99% certainty that 60% of Sonar's predictions for the remaining 396 source code components would be exact.

4.2.4.2 Results for bool

As for scalc, we first performed a first manual analysis on *bool* for each of the twenty-five source-code components. Second, we let Sonar compute its predictions for the same source code components. Table 12 presents the results of both predictions using the same convention than for scalc, in particular, discrepancies in predictions are in bold.

Table 12: Comparison of the results of a manual analysis and of Sonar for *bool*.

Pos.	Manual Results	Sonar's results that help correct manual results
1	All but 2 unsuccessful search (text/html Fail)	
2	All	
3	All	
4	All	
5	Text & Html count number of matches	
6	Text & Html find a fixed string	
7	All	
8	OR all sub-tree	
9	All Html search	
10	**All Html search**	**Except unsuccessful Html search, and count number of matches in Html file**
11	**All Except count number of matches**	**All Except count number of matches and failed search**
12	**All Html search**	**All Html search except failed**

		html search
13	All but search a fixed string	
14	All Html search	
15	All but search a fixed string	
16	None	
17	All Html search	
18	Text & Html find a fixed string	
19	**Text & Html find *n* matches**	**All except unsuccessful search, and count number of matches**
20	**All**	**All but count number of matches**
21	All text search	
22	**Text find a special match + failed text search**	**Text find a special match**
23	**All Html search**	**All Html search except count number of matches in Html file**
24	All	
25	All	

Table 12 shows that there are seven discrepancies and eighteen predictions where our first manual analysis gives the same predictions as Sonar's. As indicated in our protocol, we assume that the eighteen same predictions are exact. On the other hand, for the inconsistent predictions, we now perform a second manual analysis to determine whether Sonar's predictions are unsafe or safe and exact or inexact. Rather than providing the results for the inconsistent predictions, we give Table 13 with all twenty-five results computed after our second analysis. These results are now assumed to all be exact. We can then compare them with Sonar's. For safe and unsafe results, we also give the extra and missing software functions. Moreover, to show that a prediction from our first manual analysis has been changed by our second analysis, we italicized it. In other words, this points out when our first manual analysis was wrong. It happened in four cases, in rows 10, 19, 20, and 23.

Table 13: Estimation of Sonar's results for *bool*.

Pos.	Correct Results	Sonar's results that help correct manual results
1	All but failed text/html search	Correct
2	All	Correct

3	All	Correct
4	All	Correct
5	Text & Html count number of matches	Correct
6	Text & Html find a fixed string	Correct
7	All	Correct
8	OR all sub-tree	Correct
9	All Html search	Correct
10	*All Html search except unsuccessful Html search, and count number of matches in Html file*	Correct
11	All Except count number of matches	**UNSAFE:** Missing text/html failed search
12	All Html search	**UNSAFE:** Missing failed html search
13	All but search a fixed string	Correct
14	All Html search	Correct
15	All but search a fixed string	Correct
16	None	Correct
17	All Html search	Correct
18	Text & Html find a fixed string	Correct
19	*All except failed search, and count number of matches*	Correct
20	*All but count number of matches*	Correct
21	All text search	Correct
22	Text find a special match + failed text search	**UNSAFE:** Missing failed text search
23	*All Html search except count number of matches*	Correct
24	All	Correct
25	All	Correct

First, we note that three of Sonar's predictions are unsafe (11, 12, and 22). Second, the remaining twenty-two predictions are exact. Surprisingly, all twenty-two predictions are safe and imprecise.

The results above only provide information for 25 of the 508 source-code components exercised. However, we can use the binomial distribution to estimate what the predictions would be for the remaining 483 source-code components. In particular, we can determine the X and Y that make our hypothesis below correct.

Goal: We want to determine with more that 99% certainty that for software system

Z, Sonar computes X% of safe predictions and Y% of exact predictions when *Exercise* and *Activate* relationships are sampled with a set of system tests that satisfy our test selection criteria.

We may say that for the *bool* software and our 21 system tests that satisfy our test selection criteria

- X = 70%. We know with more than 99% certainty that 70% of Sonar's predictions for the remaining 483 source code components would be safe
- Y = 70%. We know with more than 99% certainty that 70% of Sonar's predictions for the remaining 483 source code components would be exact.

4.2.5 Conclusion of study

Although *scalc* and *bool* are very different in nature—the first is object-oriented and interactive while the second is procedural and command-line driven—the results of our study remain very similar. In both cases, safety of predictions is around 70%, and the level of correctness varies of only 10% (between 60% and 70%) between the two systems. Thus, when *Exercise* and *Activate* relationships are sampled with a set of system tests that satisfy our test selection criteria, the safety and the correctness of predictions does not vary dramatically between the two systems selected. This is encouraging. If these results repeated on several other systems, we would be able to infer that the way a system is implemented does not influence Sonar's predictions (when system tests satisfy our test selection criteria).

However, the current rate of safety (70%) and correctness (between 60 and 70%) must be improved before we can use these results as a basis for inferring new *Potentially Affect* relationships. Our future work will therefore not only focus on testing Sonar and our test selection criteria with other software systems but also on developing a technique to improve the current level of safety and of correctness of predictions. This may be done through refinement. After *Exercise* and *Activate* relationships are first sampled with a set of system tests that satisfy our test selection criteria, more system tests are selected for further refinement of the sampled *Exercise* and *Activate* relationships. Another solution is to create new test selection criteria.

A particular area that needs help from Sonar is that of large software systems. However, currently we have no way to guarantee the correctness of manual results when systems are large. For such systems, assuming the correctness of predictions is not acceptable. A promising direction is to study the help provided by Sonar's predictions instead of studying the rate of safety and correctness of Sonar's predictions. In other words, although we cannot guarantee the safety and the correctness of Sonar's results, can we determine if Sonar's predictions teach new information to the programmer? The information below shows that such studies are in fact possible, and they are likely to produce very useful results.

Our case study has a very interesting side effect. In fact, if we look back at our first and second manual predictions, we find that for systems *scalc* and *bool* four predictions manually computed during our first analysis are wrong; therefore, they were updated by our second manual analysis. In many environments, programmers can only investigate the source code using the technique used by our first manual analysis. In particular, programmers do not have the time to manually instrument the source code (with *print* statements) and execute the system to determine certain dynamic behaviors. In these environments, programmers are limited to code review assisted with text search tools (such as *grep*) in order to determine the ripple effect of a source code change on the software functionality. As the results of our case studies on *scalc* and *bool* show, results of the first manual analysis were wrong four times.

When further analyzing these results, we observe that when the first manual analyses are wrong, those of Sonar are correct. This is true for the four cases of both scalc and bool. We definitely want to find out if that trend generalizes. If it does generalize, it will indicate that when the manual analysis is difficult and the programmer has a greater risk to commit an error, Sonar has a high probability to compute the correct predictions, or at least a safe prediction. Hence, the use of Sonar with *Exercise* and *Activate* relationships sampled from system tests that satisfy our test selection criteria would be computing predictions of great assistance to programmers. This approach of studying the usefulness of Sonar's predictions seems to be a promising direction, especially for large systems.

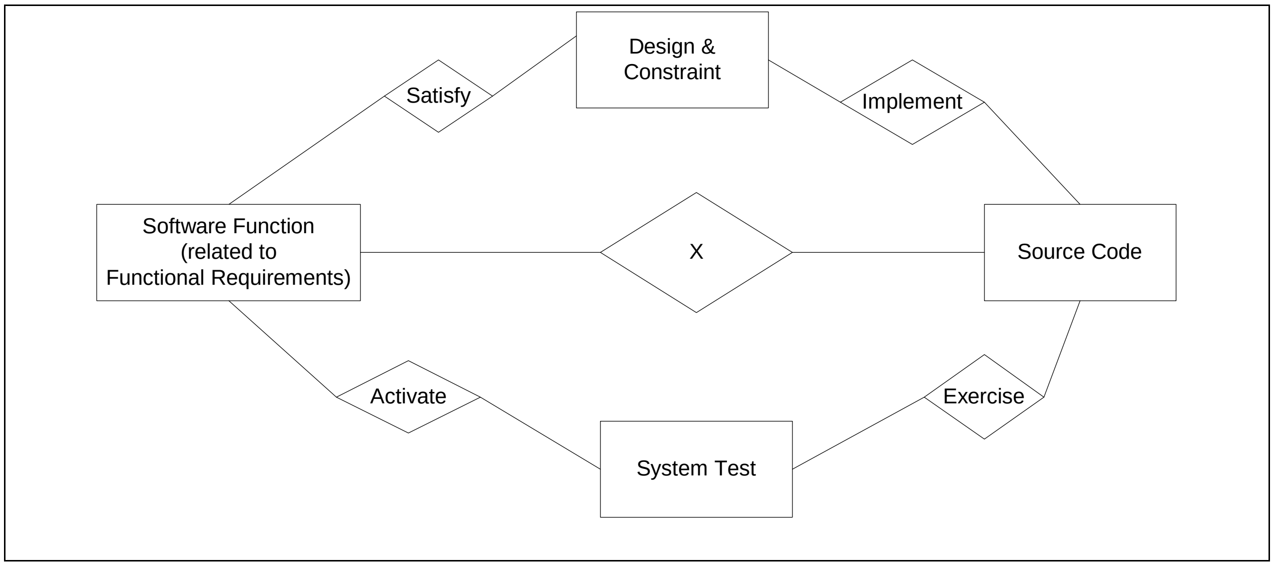


Figure 18: E-R diagram of software components and their relations.

5 Related Works

To introduce related works, we reuse the E-R diagram presented in Chapter 1. Several researchers have worked on relating software specifications to source code. From Figure 18, we see three possible ways to relate software specification to source code: (1) one direct approach, (2) two transitive approaches through design and constraints, or (3) using system tests.

Antoniol et al. propose a direct approach. They directly map the functional requirements to the source code. Their technique uses the similarity between the requirements document vocabulary and the names of identifiers in the source code in order to relate software functions (or functional requirements) to source code components [Antoniol et al. 2000]. They performed a case study on a real-world system that showed that their method traded off quite a bit of precision to get safe results. They found that to get safe predictions they had to allow for a very low level of precision (12%). In other words, all software functions affected are part of the predictions; however, only one out of eight predictions is truly affected. When they tried to improve the precision, the safety of predictions suffered dramatically. For example, when predictions are safe at 50% (half of the affected software functions are not in the prediction), they found that the precision of the prediction is at 54% (half of the software functions in a prediction are really not affected). These percentages indicate that their method does not currently produce good results on real-world systems. Moreover, their method can only study its reliability through empirical studies. That is, there is no framework to study the general theoretical reliability of their method. On the other hand, our method has enabled us to theoretically study the safety of predictions. Moreover, from our case study, we found that our method seems to be more accurate than that of Antoniol et al. However, their method has less setup costs than our method since system tests are not required. Furthermore, with their method, a programmer can query from any part of the source code, independent of whether this source code is executable or not. In our case, the programmer can only query the source code components that were exercised by a system test.

A second technique for relating software specifications to source code uses design and/or formal constraints. In Figure 18, the two relations of interest are *Satisfy* and *Implement*. Commercial companies such as Rational™ or TogetherSoft™ have pushed this approach. Gates et al. also propose a similar model where formal constraints in the form of logic rules enable inferring relationships from source code to requirements [Gates and Della-Piana 1997, Gates and Li 1998, Gates and Teller 2000]. When correctly applied, these methods are sound and give good predictions. The downside is the setup cost. These methods require the existence of a well-defined software development process where design and/or logic rules are created during the initial software development cycle and, more importantly, are maintained in subsequent cycles of development and maintenance. Such maintenance must not only update the design and/or logic rules but also update their *Satisfy* relationships with software requirements. The manual effort of maintaining *Satisfy* relationships requires tedious work. Over time, errors are likely to be introduced, which compromises the integrity of *Satisfy* relationships. Developing a method that automates computing *Satisfy* relationships may prove to be very challenging. In contrast, our method can be applied to software projects that did not start with a rigorous process requiring design and logic rules to be created. In our case, only system tests are needed to enable the application of our method. Currently, it is still more common to find companies with up-to-date test suites for their software products rather than with up-to date requirements document, design, and source code all in-sync. In any case, product and practice suggested by Rational are gaining acceptance in the software industry. Thus, in the future, our method may be combined with that of the Rational Unified Process. Our method for predicting *Potentially Affect* relationships can then help verify the integrity of *Satisfy* relationships. In particular, after a software maintenance has been tested with an instrumented version of the system, the *Potentially Affect* relationships computed by system testing can be used to point out the *Satisfy* relationships that need updating between requirements and design components.

The third method, which is ours, requires system tests. Several efforts prior to ours have used system tests to relate software functions to source code. However, they all compute relationships between software functions and source code to go from a software function to source code. In particular, they provide heuristics to locate the implementation of a particular software function in the source code. In other words, the query starts from a software function, and the result predicts the source code components that implement that software function. These methods do not answer the same question as ours. However, since they also relate software functions to source code using system tests, it is important for us to present them.

Parikh and Zvegintzov were the first to propose comparing the execution traces of system tests to find information with potential relevance to specific software maintenance [Parikh and Zvegintzov 1983]. In particular, they proposed to compare the exercise traces of system tests that activate the software functions directly related to the proposed maintenance with the exercise traces of similar system tests that do not activate the software functions related to the maintenance. Segments of source code related to the first exercised trace but not to the second are potential locations where the maintenance could take place. A programmer can start investigating the source code from these segments. Wilde and Scully, Reps et al., and Wong et al. implemented a tool to facilitate using this approach [Wilde and Scully 1995, Reps et al. 1997, Wong et al. 1999]. Wilde and Scully implemented Software Reconnaissance, and Wong et al. implemented χSuds. Both tools represent exercised traces using node profiles. Reps et al.'s technique proposed to represent exercised traces using acyclic intraprocedural paths (B-L paths) to identify Y2K related problem in the source code. Only Wilde and Scully studied issues related to the theoretical aspect of their heuristics, but to do so, they assumed the existence of an infinite number of system tests. On the other hand, we have proposed a method to compute safe results for a large category of software functions where only a finite number of system tests is required. Hence, unlike Wilde and Scully's method, our analysis remains tractable.

6 Conclusion and future directions

This research has explored a method to identify the software functions potentially affected by a change at a selected spot of the source code. This method uses system tests to infer relationships between software functions and the source code of a software system. In particular, the system tests sample *Exercise* and *Activate* relationships between the system tests and the source code components and between the system tests and the software functions, respectively. Our method then consists of joining the *Exercise* and *Activate* relationships information to detect the ripple effects from a change in the source code on software functions.

We found the conditions needed for our method to guarantee safe predictions for a large class of software functions. Some of these conditions specify the source code coverage that the system tests must achieve; in particular, all complete sets of interprocedural paths as defined by Melski and Reps (M-R paths) must be exercised. Although a finite number of system tests can achieve this coverage, no practical number of system tests can do it. However, this source code coverage criterion proposes a finite limit to our problem of safely predicting the software functions potentially affected by a source code change. Later works may use this limit as a stopping criterion for their algorithm. Since there are a finite number of complete sets of M-R paths, solutions will always be tractable. For example, we plan to develop a method where a few system tests are used to compute seeds *Exercise* and *Activate* relationships. These seeds will then be used to propagate information pertaining to software function to all complete sets of M-R paths. Actually, further research is needed to reduce the complete sets of M-R paths so that the predictions will remain safe and the level of precision will improve. In fact, when all complete sets are covered, we know that the predictions will be safe but highly imprecise.

We developed Sonar, a prototype tool that implements our method. Our case studies used Sonar on small real systems. For these studies, we created a new set of test selection criteria that were always satisfied by a few system tests. These criteria are generic; therefore, more case studies on different software systems can be done to further test our test selection criteria. Although our results are better than the method of Antoniol et al., they are still not good enough to be used as seeds by a propagation technique such as the one described in the previous paragraph. In fact, seeds relationships may only be used if we are highly confident that the information propagated is safe and fairly precise. Currently, our two case studies have shown that predictions are safe only 70% of the time. The percentage of safe predictions would need to be in the upper nineties in order to qualify as good seeds.

Although our results are not good enough for programmers to rely only on them, our case studies have shown that programmers' manual analyses would benefit from our predictions. In particular, a side effect of our studies has illustrated that each time the programmer made an error in the manual predictions Sonar computed safe predictions. If this tendency generalizes, it would definitely show that our method is useful to programmers when they are manually performing difficult crucial analyses on how a change at a particular spot of the source code impacts a software application's functionality.

We now develop three types of future works. The first task is to improve Sonar. We intend to tailor a program profiling technique that helps compute the *Exercise* relationships between a partial execution of a system test and a partial exercise trace. Indeed, *Exercise* now relates a system test to its complete exercise trace. In the case of interactive programs, a system test often activates a sequence of several unrelated software functions. It may be useful to partition the execution of such a system test and only relate each partial execution to its corresponding partial exercise trace. Such partitioning will likely improve the precision of Sonar's results. Creating a relation between the partial execution of system tests and partial exercise traces would be fairly straightforward for systems that run as single-process, but, in the case of multiprocess or multithreaded programs, it becomes much harder to create relationships between partial execution. Since interactive programs with GUI often run in multiple threads, it would be crucial for the new profiling technique to handle such cases. Annotated grammar used to compute *Activate* relationships may assist the partitioning of a system test into segments that correspond to its unrelated software functions. Recursion in the rules of a feature grammar such as the one of the bank ATM presented in Chapter 2 may help indicate the cycles in source code that determine the break points, that is, where a software function terminated and another started executing.

We are also interested in implementing Sonar for analyzing Java programs. At the moment, it is limited to the study of C and C++ programs. In the case of Java programs, we could use the built-in profiling interface JVMPI to help compute *Exercise* relationships.

The second task is to improve on our current method for computing predictions by not only using dynamic analysis but also using static analysis. We already mentioned this direction earlier in this chapter when describing the use of seeds relationships and a propagation technique. In particular, the goal is to dynamically obtain a few seeds relationships that compute predictions with a high degree of safety and precision for a limited number of source code components. In parallel, we can develop heuristics to propagate the information of seeds relationships throughout the rest of the source code based on static analysis.

The third future task consists of conducting more experiments with Sonar to verify if our claims generalize on large systems. In particular, we would like to collaborate with the software industry and verify whether our claims remain true on real-world systems. However, when studying the application of our method on a large project, it may not be feasible to determine whether predictions are safe and precise. A more relevant question is: are predictions providing new information to programmers? And is the new information important to the point that it will avoid the introduction of bugs in a future release of a software system? In parallel, we plan to perform a comparative study on the effort required for particular maintenance in relation to the coupling between software functions in source code. This could then define software metrics to be incorporated in Sonar.

Our desire is to help programmers in the process of modifying a program. So far, programmers determine the ripple effect of source code modification on software function using ad hoc techniques made of code and documentation review and, when possible, of exercise traces review. Currently, we know that our method may not be capable of guaranteeing safe and precise predictions when using only a reasonable set of system tests. However, we find it important that we can bring new information to the table. In particular, if the new information is important to the point that it will avoid the introduction of bugs in a future release of a software system then our method is worth applying.

7 References

[Agrawal 1999] H. Agrawal, “Efficient coverage testing using global dominator graphs,” *SIGSOFT Software Engineering Notes*, vol. 24, 1999, pp. 11-20.

[Agrawal, et al. 1993] H. Agrawal, R. A. DeMillo, and E. H. Spafford, “Debugging with dynamic slicing and backtracking,” *Software - Practice and Experience*, vol. 23, 1993, pp. 589-616.

[Aho et al. 1986] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, Principles, Techniques, and Tools.* Addison Wesley, January 1986.

[Albrecht and Gaffney 1983] A. J. Albrecht and J. E. Gaffney, “Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation,” *IEEE Trans. Software Eng.*, November 1983, pp. 639-648.

[Antoniol et al. 2000] G. Antoniol, G. Camfora, A. De Lucia, G. Casazza, and E. Merlo, “Tracing Object-Oriented Code into Functional Requirements”, in *Proc. of the 8th International Workshop on Program Comprehension (IWPC’00)*, Limerik, Ireland, June 10-11, 2000, pp. 79-86.

[Ball and Larus 1996] T. Ball and J.R. Larus, “Efficient Path Profiling,” in *Proc. of 29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-29)*, Paris, France, December 2-4, 1996, pp. 46-57.

[Ball 1998] T. Ball, “On the limit of control flow analysis for regression test selection,” in *Proc. of the ACM/SIGSOFT International Symposium on Software Testing and Analysis*, Clearwater Beach, FL, March 2-5, 1998, pp. 134-142.

[Balzer 1969] R. M. Balzer, "EXDAMS—Extensible Debugging and Monitoring System," in *AFIPS Proceedings of the Spring Joint Computer Conference 34*, Washington, D.C., 1969, pp. 125-134.

[Beck 1999] Ken Beck, *Extreme Programming Explained: Embrace Change*. Reading, MA: Addison Wesley, October 1999.

[Binkley 1995] D. Binkley, "Reducing the cost of regression testing by semantics guided test case selection," in *Proc. of the Conference on Software Maintenance 1995 (CSM95)*, Opio (Nice), France, October 17-20, 1995, pp. 251-260.

[Camuffo, et al. 1990] M. Camuffo, M. Maiocchi, and M. Morselli, "Automatic software test generation," *Information and Software Technology*, vol. 32, 1990, pp. 337-346.

[Celentano, et al. 1980] A. Celentano, S. C. Reghezzi, P. D. Vigna, C. Ghezzi, G. Gramata, and F. Savoretti, "Compiler Testing using a Sentence Generator," *Software - Practice and Experience*, vol. 10, 1980, pp. 987-918.

[Cusumano and Selby 1995] Michael A. Cusumano and Richard W. Selby. *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*. New York, NY: Simon & Schuster, December 1998.

[Davis 1993] Alan M. Davis, *Software Requirements: Objects, Functions, and States*. Upper Saddle River:NJ, Prentice Hall, 1993.

[Deprez and Lakhotia 2000] J-C. Deprez and A. Lakhotia, "A formalism to automate mapping from features to code," in *Proc. of the 8th International Workshop on Program Comprehension 2000 (IWPC2000)*, Limerick, Ireland, June 10-11, 2000, pp. 69-78.

[Erdem, et al. 1998] A. Erdem, W. L. Johnson, and S. Marella, "Task oriented software understanding," in *Proc. of the Thirteenth International Conference on Automated Software Engineering*, Honolulu, HI, October 13-16, 1998, pp. 230-239.

[Ernst et al. 1997] Michael Ernst, Greg J. Badros, and David Notkin, "An Empirical Analysis of C Preprocessor Use, " technical report UW-CSE-97-04-06, University of Washington, Seattle, WA, April 22, 1997.

[Fischer 1977] K. F. Fischer, "A test case selection method for the validation of software maintenance modifications," in *Proc. of Computer Software and Applications 1977 (COMPSAC'77)*, New York, NY, 1977, pp. 421-426.

[Fischer, et al. 1981] K. F. Fischer, F. Raji, and A. Chruscicki, "A methodology for retesting modified software," in *Proc. of the National Telecommunications Conference B-6-3*, November, 1981, pp. 1-6.

[Gates and Della-Piana 1997] A. Gates and C. Della-Piana, "The identification of integrity constraints in requirements for context monitoring," in *Proc. of the 1997 IEEE International Conference and Workshop on Engineering of Computer-Based Systems (ECBS'97)*, Monterey, CA, March 24-28, 1997, pp. 498-505.

[Gates and Li 1998] A. Gates and S. Li, "Software Faults and their Detection through DynaMICs," in *Proc. of the International Association of Science and Technology for Development (IASTED) Software Engineering Conference*, Las Vegas, NV, October 28-31, 1998, pp. 323-327.

[Gates and Teller 2000] A. Q. Gates and P. J. Teller, “DynaMICs: An Automated and Independent Software-Fault Detection Approach, ” in *Proc. of the Fourth IEEE International High Assurance Systems Engineering Symposium*, Washington, D.C., November 1999, pp. 11-19.

[Griswold et al. 1996] W.G. Griswold, D.C. Atkinson, and C. McCurdy. “Fast, flexible syntactic pattern matching and processing, ” In *Proc. 4th Workshop on Program Comprehension*, Berlin, Germany, March 28-31, 1996, pp. 144-153.

[Hanson 1978] D. R. Hanson, “Event associations in SNOBOL4 for program debugging,” *Software - Practice and Experience*, vol. 8, 1978, pp. 115-129.

[Harrold and Soffa 1989] M. J. Harrold and M. L. Soffa, “Interprocedural data flow testing,” in *Proc. of the Third Testing, Analysis, and Verification Symposium*, Key West, FL, December 13-15, 1989, pp. 158-167.

[IEEE 1983] IEEE Standard for Software Test Documentation, ANSI/IEEE STD 829, 1983.

[Melski and Reps 1998] D. Melski and T. Reps, “Interprocedural path profiling, ” TR-1382, Computer Sciences Department, University of Wisconsin, Madison, WI, September 1998.

[Melski 2002] David Melski, “Interprocedural path profiling and the interprocedural express-lane transformation,” Ph.D. dissertation, Computer Sciences Department, University of Wisconsin, Madison, WI, 2002.

[Parikh and Zvegintzov 1983] G. Parikh and N. Zvegintzov, *Tutorial on Software Maintenance*. Silver Spring, MD: Computer Society Press, 1983.

[Purdom 1972] P. Purdom, “A Sentence Generator for Testing Parsers,” *BIT*, vol. 12, 1972, pp. 366-375.

[Reps, et al. 1997] T. Reps, T. Ball, T. M. Das, and J. Larus, “The use of program profiling for software maintenance with applications to the Year 2000 Problem,” *Lecture Notes in Computer Science*, vol. 1301, 1997, pp. 432-449.

[Rothermel and Harrold 1997] G. Rothermel and M. J. Harrold, “A safe, efficient regression test selection technique,” *ACM Transaction on Software Engineering and Methodology*, vol. 2, 1997, pp. 173-210.

[Sommerville 1992] I. Sommerville, *Software Engineering, 4th edition*. Reading, MA: Addison Wesley, 1992.

[Spadafora and Bazzichi 1982] I. Spadafora and F. Bazzichi, “An automatic generator for testing compiler,” *IEEE Transaction on Software Engineering*, vol. 8, 1982, pp. 343-353.

[Tolmach and Appel 1990] A. P. Tolmach and A. W. Appel, “Debugging standard ML without reverse engineering,” in *Proc. of the 1990 ACM Conference on LISP and Functional Programming*, Nice, France, June 27-29, 1990, pp. 1-12.

[Wilde and Gust 1992] N. Wilde and T. Gust, “Locating User Functionality in Old Code,” in *Proc. of Conference on Software Maintenance*, Orlando, FL, November 9-12, 1992, pp. 200-205.

[Wilde and Scully 1995] N. Wilde and M. C. Scully, “Software Reconnaissance: Mapping Program Features to Code,” *Software Maintenance: Research and Practice*, vol. 7, 1995, pp. 49-62.

[Wong, et al. 1999] W. E. Wong, S. S. Gokhale, J. R. Horgan, and K. S. Trivedi, "Locating program features using execution slices," in *Proc. of the Second IEEE Symposium on Application-Specific Systems and Software Engineering Technology*, Richardson, TX, March 24-27, 1999, pp. 194-203.

[Wong, et al. 1997] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice," in *Proc. of the Eighth IEEE International Symposium on Software Reliability Engineering*, Albuquerque, NM, November 2-5, 1997, pp. 522-528.

[χAtac] Telcordia Technologies Inc. *Telcordia Software Visualization and Analysis Toolsuite (χSuds)—User's Manual, first edition*. Morristown, NJ: Telcordia Technologies Inc., 1998. Available at http://xsuds.argreenhouse.com/html-man/xsudsTOC.html, (χAtac belongs to the χSuds toolsuite).

Deprez, Jean-Christophe. Bachelor of Science, University of Southwestern Louisiana, 1994; Master of Science, University of Southwestern Louisiana, 1997; Doctor of Philosophy, University of Louisiana at Lafayette, Spring 2003
Major: Computer Science
Title of Dissertation: Detecting Ripple Effects of Program Modifications on a Software System's Functionality
Dissertation Director: Dr. Arun Lakhotia
Pages in Dissertation: 141; Words in Abstract: 298

ABSTRACT

When changing a line of source code, a programmer needs to know how changing that line may affect the end-user functionality of the software system. In this dissertation, we explore a method that uses system tests to relate software functions (units of software functionality) to source code. This method can be used to predict the software functions potentially affected by a change at a particular spot in the source code. The quality of a prediction is measured in terms of its safety and its precision. These two attributes are respectively addressed by answering the following questions: Are all potentially affected software functions predicted, and are all software functions predicted potentially affected?

We define a source code coverage criterion in terms of sets of inter-procedural paths. When a system test that satisfies this criterion is used, our method guarantees safe predictions for a large class of software functions. For most systems achieving such source code coverage may require an exponential number of system tests. Moreover, the precision of predictions is not guaranteed. Consequently, we create a new set of test selection criteria on the basis that all these new criteria must always be satisfied by a small number of system tests. Case studies on two software systems, namely *scalc* and *bool*, show that sets of system tests that satisfy our new criteria enable our method to compute safe predictions 70% of the time and safe and precise predictions between 60–70% of the time.

These results are not at a level where our method would supersede a programmer’s manual analysis. However, they complement manual predictions by improving a programmer’s confidence in the result of her/his manual analysis. Incidentally, during our two case studies, we observed that our method always corrected the programmer when he made a wrong manual prediction.

BIOGRAPHY

Jean-Christophe Deprez received his Bachelor’s degree in Computer Science from the University of Southwestern Louisiana in 1994. After working at Pfizer for a year, he came back to Louisiana for his graduate studies where he joined the Center for Advanced Computer Studies (CACS) under the Graduate School Fellowship program. He first focused his studies on using static analyses of programs to assist software reengineering. This lead to his master thesis: “A Context Sensitive Transformation for Restructuring Programs.” He obtained a Master of Science in Computer Science from the University of Southwestern Louisiana in 1997. After a six month sabbatical, he returned to Louisiana for his doctoral research. He studied dynamic program analyses and their use in assisting program comprehension. Since Fall 2001, he has been an Assistant Professor in the Computer Science and Information System Department of Pace University in New York.