A Unifying Framework Supporting the Analysis and Development of Safe Regression Test Selection Techniques

John Bible
Department of Computer Science
Oregon State University
Corvallis, OR 97331
bible@cs.orst.edu

Gregg Rothermel
Department of Computer Science
Oregon State University
Corvallis, OR 97331
grother@cs.orst.edu

January 8, 2000

Abstract
Safe regression test selection (RTS) techniques let software testers reduce the number of test cases that need to be run to revalidate new versions of software, while ensuring that no fault-revealing test case (in the existing test suite) is excluded. Most previous work on safe regression test selection has focused on specific safe RTS algorithms, rather than addressing the theoretical foundations of safe RTS techniques in general. In this paper, we present a unifying framework for safe RTS that supports the analysis and development of safe RTS techniques. We show that every safe RTS technique is founded on a regression bias, and we show how one can prove an RTS technique safe for a set of programs and testing processes by eliciting the bias from the technique and then proving that the bias holds or does not hold for that set of programs and processes. We provide two general models for safe RTS techniques that can be used as templates in proofs of safety. The first model actually contains two formulations, both defined in terms of finite automata. This first model is inefficient; it does, however, provide the most general, powerful, safe RTS algorithm possible, given some simple assumptions. We then define a generalized safe RTS algorithm that is only marginally less powerful than the finite automata method, and that provides a template that simplifies the development of safe RTS algorithms. Finally, to illustrate the application of our framework, we use it to analyze several existing safe RTS algorithms and their biases, and to develop a new safe RTS algorithm for programs written in spreadsheet languages.

1 Introduction

The cost of maintaining quality in new versions of software dominates the cost of software maintenance. Many maintenance activities contribute to software quality; current practice emphasizes the use of regression testing. When regression testing requires the creation of test suites, it can be particularly expensive. However, in the maintenance phase (unlike the development phase), test suites from previous validation phases may already exist, and we can reuse those suites. But in the attempt to reuse existing test suites, two problems arise. First, should we reuse all, or only part of, the test suites? Second, what new test cases should we develop to satisfy new specification requirements for the modified software? The first problem is known as the regression test selection problem; the second problem is the test suite augmentation problem. Both of these problems are important; however, in this paper we focus on the regression test selection (RTS) problem.

To solve the RTS problem, software testers must first decide whether or not to use all of the existing test suite. Running all of the test cases in an existing test suite can consume an inordinate amount of time: for example, one of our industrial collaborators reports that for one of its products — containing only about 20,000 lines of code — running the entire test suite requires seven weeks. On the other hand, omitting test cases from the test suite can potentially result in the exclusion of a fault-revealing test case. If the testing
requirements do not forbid this risk, a regression test selection (RTS) technique (e.g. [2, 5, 6, 8, 11, 13, 15, 18, 19, 25, 29, 31, 32]) can be employed; such techniques select a subset of the test suite using heuristics or analytic approaches.

Different RTS techniques satisfy different demands. The simplest RTS techniques select test cases randomly, whereas more complex techniques typically base their selection on code analysis and test execution profiles. These different RTS techniques produce test suites that have different fault-detection capabilities, and generate test suites that are nonequivalent in terms of meeting customer requirements. In this paper, we focus on RTS techniques that are required to be safe [5, 11, 18, 26, 29]. Safe RTS techniques select (under certain conditions) every test case from the original test suite that can reveal a fault in the modified program. For maintainers of safety-critical applications, such conservatism can be essential; however, even in less critical domains, incurring the extra costs of employing safe rather than non-safe RTS techniques might be preferable to releasing faulty software.

Previous research has produced several algorithms for safe RTS. These algorithms all employ similar methods in that they compare source code or program representations for a program and its modified version to locate lexical differences, and select test cases associated with those differences. For example, the Rothermel/Harrold tool DejaVu [26], the Vokolos/Frankl tool Pythia [29], and the family of techniques by Ball [5] use direct lexical comparisons of source code statements, relying on control flow information to locate test cases that could reach code that has changed. An approach by Laski and Szemer reduces the control flow graphs for two versions of a program to a form in which modified regions of code can be identified [18]. The TestTube approach, created by Chen, Rosenblum, and Vo [11], uses checksum comparisons on the text of entities in the global namespace: functions and variables, for instance.

As the previous descriptions indicate, most safe RTS techniques also rely, implicitly or explicitly, on control flow information to identify changed code or test cases that must be re-executed. (The exception is TestTube, whose simple analytic method operates at such a coarse level that control flow information is neither stored nor required). Partially, this is because existing safe RTS techniques target imperative languages. For functional languages such as Haskell and pure Lisp, for logic languages such as Prolog, and for declarative languages such as those used in spreadsheets, however, control flow is not as significant a component. In fact, constructing techniques based on control flow in these cases may be impossible, or may mask opportunities for being more efficient.

Further, existing safe RTS techniques all require that code execute deterministically. Thus, even though distributed and parallel imperative programs rely locally on control flow, they cannot easily be operated upon safely by existing techniques. Also, existing safe RTS techniques do not account for nondeterministic response times or nondeterministic event arrival times: thus, real-time and event-driven programs often are difficult to accommodate using these techniques. We believe that these limitations are partially the result of the lack of a general framework for safe RTS – a framework that specifies what safe RTS means with respect to these different types of programs.

So far, little has been done to establish a set of requirements to govern, or a model with which to evaluate, safe RTS techniques in general. Rothermel and Harrold [25] provide a framework for comparing and analyzing regression test selection techniques [25], but that framework does not specifically address the foundations of safe RTS. Ball [5] addresses the theoretical foundations of control-flow-graph-based safe RTS techniques such as DejaVu and shows that the process employed by these techniques is functionally
equivalent to the process of taking the intersection of deterministic finite automata. The primary foci of [5], however, are control-flow-based techniques and issues of precision (the extent to which techniques succeed in omitting unnecessary test cases).

The lack of a foundation for safe RTS makes proofs of safety for techniques difficult to achieve, yet such proofs are essential if these techniques are to be deemed acceptable for use in validating safety-critical software. The lack of a foundation also obscures the unique needs of different varieties of programs and testing situations to which safe RTS might be applied.

In this paper, we present a unifying framework for safe RTS algorithms, that supports the analysis, comparison, and creation of safe RTS algorithms generally. This framework defines the properties that an RTS algorithm must possess in order to claim safety. In particular, the framework establishes the concept of a regression bias — the set of assumptions upon which the safety of an RTS technique is based — and defines a set of primitive entities out of which a model of a safe RTS technique, and a proof of safety of that technique, can be built, given that the bias is satisfied by the program and testing process.

The framework also includes two general models for safe RTS techniques that can be used as templates in proofs of safety. We first define two formulations of a model based on finite automata; these two formulations provide a theoretical basis for safe RTS, although they are not particularly efficient. We then present a generalized safe RTS algorithm that generates a signature graph — a graph that guarantees that a test case can be safely excluded if the trace of that test case for the original program $P$ remains entirely within the graph. This algorithm can be used as a template for new techniques that implement the generic functions present in the algorithm, or it can be used to facilitate the analysis and comparison of safe RTS techniques. We also present an extension to this general algorithm that is equivalent in precision to the second formulation of the finite automata model, but more efficient at analyzing program differences.

Finally, we illustrate the use of our framework. We first show how the framework can be used to provide a proof of safety for DejaVu, and sketching its use in providing similar proofs for Pythia and TestTube. We then show how our framework can be used to develop a new safe RTS technique for use in testing spreadsheet programs; this application also demonstrates how the technique can be applied to programs that are not reliant on control flow for proper evaluation.

The remainder of the paper is organized as follows. Section 2 provides background material required for the subsequent discussion. Section 3 describes the semantic framework for safe RTS, and presents both formulations of the first model of safe RTS. Section 4 describes the generalized safe RTS algorithm and its template functions, and presents its more precise extension. Section 5 presents the applications of the framework, including its use in the analysis and proof of DejaVu and other techniques, and in the creation of a new instantiation of the generalized algorithm for use with spreadsheets. Section 6 presents conclusions.

2 Safe Regression Test Selection

Unlike software under initial development, software under maintenance may have already been partially validated, through validation of previous versions. New functionality, of course, has not been subjected to testing, and thus requires the creation of new test cases. Old functionality, however, may already have been exercised with existing test suites; testers can revalidate this functionality and examine whether changes have introduced new faults into old code by re-executing these existing test suites. Thus, the retesting of
modified software involves two distinct problems:

1. The problem of validating changed functional and non-functional requirements of the new version of the program,

2. The problem of verifying that changes have not introduced faults detectable by the original test suites.

The first problem requires validation of new and enhanced system components, and this requires the development of new test cases: we call this the test suite augmentation problem. Changes to code can also require development of new test cases, however, when no functional changes are made. In particular, customer requirements (such as code coverage requirements) for the testing process that were previously met might, following modifications, fail to be met, requiring test case creation. In either case, this task resembles the original development-phase testing process rather than the regression test selection process that this paper targets, so we do not address it here.

The second problem requires revalidation of old non-obsolete functionality and requirements using existing test cases: this is the regression test selection problem and is the focus of this paper.

Throughout the rest of this paper we use $P$ to denote a program, $P'$ to denote a modified version of $P$, and $T$ to denote an existing test suite for $P$. The regression test selection problem, then, involves selecting a subset $T'$ of $T$ for use in revalidating $P'$. Not all test cases in $T$ necessarily apply to $P'$: changes in functionality, for instance, might make some test cases obsolete. An obsolete test case is any test case $t$ whose behavior in $P'$ is specified to be different from the correct behavior of $t$ in $P$ [22]. If we remove all such test cases from $T$, however, we can then execute the remaining test cases in $T$ on $P'$ and thereby (partially) revalidate $P'$. Reusing all non-obsolete test cases in this manner is known as the retest-all approach. This approach does offer advantages to testers who must completely re-verify $P'$: while testers still pay the cost of test case execution, they do not pay the cost of test case development.

Of course, even though the retest-all technique employs only non-obsolete test cases, many of those test cases may exercise code that is unaffected by the changes that generated $P'$. The retest-all approach can involve unnecessary effort as it requires that we exercise each release with the entire test suite even if the changes in that release were insignificant. RTS techniques address this problem by selecting only a subset of the test suite to rerun, eliminating other test cases. In this paper, we are primarily interested in safe RTS techniques, which, informally, eliminate only test cases that are provably not able to reveal faults.

Before we can provide a more precise definition for the safety of an RTS technique, we require several preliminary definitions. The first of these, a definition of deterministically fault revealing, is fundamental to safe RTS as we define it in this paper.

**Definition 1. Deterministically Fault Revealing (DFR):** Given a program $P$ and a test suite $T$, we say that $T$ is deterministically fault revealing (DFR) for $P$ if, for each test case $t$ in $T$, $t$ exhibits identical failures on every execution of $P$.

Note that for $T$ to be DFR for $P$, it is not necessary that $T$ cause $P$ to exhibit deterministic behavior. A test suite can be DFR for $P$ even though $P$ exhibits random behavior; faulty behavior, however, must occur deterministically or be deterministically detectable.
Testers frequently attempt to make program behavior deterministic for the testing scenario.\(^1\) Part of the justification for doing this is to ensure that failures occur deterministically (ensuring DFR). However, there are many other important reasons for ensuring deterministic testing. For instance, failing to ensure determinism can make fault localization impossible: it is difficult to correct a fault that cannot be reproduced. Further, nondeterministic behavior under test can have disturbing effects on test quality measures at regression test time. For example, if a test case causes nondeterministic program behavior, it may exercise different branches than those it is designed to exercise, and thereby fail to cover components that it is required to cover, and fail to achieve the quality goals required of it. Further, suppose that we cannot ensure that a test suite is deterministically fault revealing: in that case, not even the retest-all technique can ensure that faults are deterministically revealed by that test suite. Thus, it is not clear what conclusions we can draw about program quality relative to the regression test suite using any form of test reuse if we cannot ensure that test suites are DFR.

Therefore, in this paper, we assume that all safe RTS techniques implicitly require test suites to be DFR for programs under test.\(^2\) As we discuss later, this assumption merely constrains the set of programs and testing methodologies over which a particular RTS technique is safe.

If a test suite has been run on \(P\) and exhibits no failures, testers often make an additional assumption related to DFR: \(P\) is correct for \(T\). If \(T\) is DFR for \(P\), and \(T\) exposed no faults in \(P\), then \(T\) is unable to expose faults in \(P\); thus, \(P\) is correct for \(T\).

\textbf{Definition 2.} \(P\) Correct for \(T\): Test suite \(T\) is incapable of exposing faults in \(P\).

Regardless of the assumptions underlying the regression testing process, however, an important goal of that process is to identify the fault-revealing subset of test suite \(T\).

\textbf{Definition 3.} Fault-Revealing Subset: Given a test suite \(T\) and program \(P\) such that \(T\) is DFR for \(P\), there exists a subset of \(T\), \(T_{FR}\), such that \(T_{FR}\) contains all and only the test cases in \(T\) that will reveal faults in \(P\) when executed on \(P\).

There is no algorithm to precisely identify, in general, the fault revealing subset of a test suite \([25]\); thus, safe RTS algorithms attempt to identify a superset of this subset. This prompts the following definitions of metrics for use in evaluating selected subsets of \(T\).\(^3\)

\textbf{Definition 4.} Inclusiveness: Given test suite \(T\) with fault revealing subset \(T_{FR}\), and \(T' \subseteq T\) where \(T'\) contains \(m\) fault-revealing test cases, the inclusiveness of \(T'\) is \(100 \times \frac{m}{|T_{FR}|}\)\(^\%\), where if \(T_{FR} = 0\), inclusiveness is defined to be 100\%.

---

\(^1\)A notable exception is the explicit nondeterministic testing of concurrent programs (e.g., \([10]\)).

\(^2\)Although we do not examine the possibility in this paper, it is possible that RTS techniques might operate safely on nondeterministically fault revealing test suites in certain circumstances. For example, it might be the case that \(T\) is not DFR, but that only some subset of \(T\) contains potentially fault-revealing test cases. In this case, we might define a subset of \(T\) called \(T_{NDFR}\), or the non-deterministic fault-revealing test suite. A similar analysis to the one presented in this paper might be made for that testing scenario, although the modifications would not be trivial. In particular, while the validating power of a DFR test suite is probabilistic in one dimension — in essence, by sampling the behaviors of \(P\) with \(T\) — the non-deterministic case would be probabilistic in two dimensions — not only would it sample \(P\) with \(T\), but the test cases in \(T\) would themselves be probabilistic.

\(^3\)Definitions 4 through 6 are similar to definitions presented in Reference \([25]\). Those definitions, however, viewed precision and inclusiveness as qualities inherent to a RTS technique, whereas here we use precision and inclusiveness with respect to a particular test suite. We have chosen the alternative definitions because RTS techniques can have profoundly different results on different types of subject programs, as we shall discuss further in Section 3.
Definition 5. Precision: Given test suite $T$ with fault revealing subset $T_{FR}$, and $T' \subseteq T$, the precision of $T'$ is $100 \times \frac{|T_{FR} \cap T'|}{|T'|} \%$, where if $T_{FR} = 0$, precision is not well-defined unless $T' = 0$, in which case precision is 100%.

Definition 6. Safety: A safe RTS technique produces test suite subsets that are 100% inclusive with respect to the base test suite $T$.

Of course, a safe RTS technique is worthwhile only when the cost $E$ of running the entire test suite exceeds the cost $A$ of running the analysis techniques plus the cost $S$ of running and validating the results of the selected test cases [19]. Or, to phrase this relation equationally:

$$A + S \leq E$$

If this property does not hold, then running a safe RTS algorithm is an inefficient allocation of resources. Thus, the precision of a safe RTS algorithm is an important factor, because it affects both the size of the selected test suite and the cost of the analysis. How one accounts for cost is critical to the above relation: for example, time spent by humans on validation may be valued more than time spent by machines on analysis.

3 A Semantic Framework for Safe RTS Algorithms

3.1 Behavioral Equivalence And Regression Biases

Safe RTS techniques, like non-safe RTS techniques, select a subset of the test cases in an existing test suite. Unlike non-safe techniques, however, safe techniques act as proofs that the excluded test cases cannot reveal faults in a given modified version of a program. In particular, safe RTS techniques attempt to prove test cases fault-behavioral equivalent over a program and a subsequent version of the program. Of course, providing such proofs over an unrestricted population of programs and testing methodologies is not possible. Instead, safe RTS techniques specialize in particular types of programs and testing methodologies: for example, deterministic programs written in C, with test cases that terminated and revealed only correct outputs on their previous execution on $P$ [26].

The situations in which safe RTS techniques apply are further refined by the use of simplifying assumptions about the environment of program execution: for example, a particular technique might require that the same compiler generate the code for $P$ and $P'$, and that the test suite $T$ execute in the same operating system for $P$ as for $P'$. In addition, requirements specifications often give leeway in judging software correct: for instance, response times might have some bound, and if a technique can show that a test case does not cause response time to exceed that bound, that test case can be assumed fault-behavioral equivalent for $P$ and $P'$. Appropriate assumptions about environment, testing methodology, and program behavior let RTS techniques be both safe and efficient, provided the assumptions are not violated. Such assumptions constitute the regression bias of a safe RTS technique.

Although we have not yet formally defined "regression bias", our informal discussion of the notion so far places it as a set of assumptions concerning not only the program code itself, but also the environment in which the the program is regression tested, and the test suites and testing methodology used to test it. These three aspects constitute a gestalt of program behavior under test, and safe RTS techniques depend on all of

---

4For a more detailed consideration of the issues involved in analyzing the economic trade-offs of RTS techniques see [14].
these aspects. However, constantly referring to the program and environment and test suite proves clumsy, and introducing a new definition seems unnecessary. Thus, in the following, we use the term “program” to refer to both code and significant influences in the external computing environment. Oftentimes, safe RTS techniques will factor out external environmental influences, in which case “program” refers only to code, but the distinction we make here points out that such a factorization is not always safe. Test suites, however, remain separate from this usage of “program”: while separating the terms adds verbiage, merging them endangers the fundamental meaning of the term program.

Given this clarification, we present precise definitions of fault-behavioral equivalence and regression biases:

**Definition 7. Fault-Behavioral Equivalence:** Given test case $t$ and programs $P$ and $P'$, $P$ and $P'$ are fault-behavioral equivalent over $t$ if the execution of $t$ on $P$ and $P'$ produces either no failures or identical failures.

**Definition 8. Regression Bias:** The set of assumptions on which a safe RTS technique constructs implicit *deductive* proofs that a subset of a test suite $T$ contains all fault-revealing test cases in $T$. (Such a bias holds for a subset of all possible programs and test suites.)

**Definition 9. Biased Fault-Behavioral Equivalence:** Given test case $t$, programs $P$ and $P'$, and regression bias $R$, $P$ and $P'$ are said to be $R$-Fault-Behaviorally Equivalent over $t$ if $R$ holding implies that $P$ and $P'$ are fault-behavioral equivalent over $t$.

In regression testing, we want to prove $P$ and $P'$ fault-behavioral equivalent over existing test suites (after obsolete test cases have been removed from the test suites). To establish the safety of an RTS technique, we must show that the technique proves $P$ and $P'$ $R$-fault-behavioral equivalent for some regression bias $R$, where $R$ is actually satisfied in the testing environment. This is a rather strict requirement in that it requires the bias $R$ to always hold in that environment; of course, if the tester can distinguish a subset $T_F$ of $T$ for which the bias fails, the tester can apply the RTS technique to $T - T_F$, and then add all test cases in $T_F$ to the selected set $T'$.

Further, while we are interested in this paper in techniques with regression biases that hold 100% of the time on programs within their intended range of application, some testers might also choose to accept “virtually safe” RTS techniques that fail on rarely occurring subject programs, where failure might mean $k\%$ inclusiveness. In this paper, we will not consider such techniques: we require 100% inclusiveness. Much of the framework presented in this paper, however, would be applicable to such virtually safe techniques.

We have defined a regression bias as a set of assumptions; this yields a set of characteristics we can use to partition the space of all programs. However, regression biases can also be viewed as sets of programs: the sets over which the assumptions hold. Thus, we shall sometimes use regression biases to describe sets of assumptions and sometimes to describe sets of programs. The context will make our specific usage clear.

Regression biases impose a general-to-specific ordering over the space $S$ of all safe RTS techniques. Each safe RTS technique relies on a bias that partitions the space $\mathcal{P}$ of programs and test suites into programs and test suites to which the bias applies, and programs and test suites to which it does not. Given that, we can define a metric for regression biases that measures the extent of the space $\mathcal{P}$ covered by the bias.

Using this general-to-specific metric, we can structure the space $S$ of safe RTS techniques with an ordering $\mathcal{R}$ over techniques. In a directed graph of ordering $\mathcal{R}$, safe RTS technique $A$ has a directed edge to safe RTS
technique $B$ if the regression bias of $B$ contains the regression bias of $A$ (in other words, all programs in $B$’s partition belong to $A$’s partition). With this ordering, we can characterize the relative strengths of different techniques over the total space of programs by examining the extents of the techniques’ underlying biases.

Figure 1 illustrates this notion. The safe RTS techniques $A$, $B$, and $C$ represented in the figure are techniques in the space $S$ of all safe RTS techniques. On the left in the figure, the space $P$ of all possible programs is shown, with RTS techniques $A$, $B$, and $C$ shown as subsets over this space; this illustrates the sets of programs over which the regression biases for these techniques apply. On the right in the figure, the space $S$ of safe RTS techniques is shown, with the general-to-specific ordering among $A$, $B$, and $C$ imposed. The figure shows that $A$ is more general than $B$, and that $C$ is more general than $B$, but that neither $A$ nor $C$ is more general than the other, even though the subsets of $P$ for which the two are safe overlap.

The most general regression biases under this general-to-specific ordering always hold, and thus always allow their corresponding techniques to produce safe subsets. The bias underlying the retest-all approach is one instance of this most general bias: the null-restriction bias. The null-restriction bias is by definition true for all programs. Since the retest-all approach selects all tests all of the time, it is guaranteed to always be 100% inclusive for all programs.

The most specific regression biases — which we term null-selection biases — hold only when no changes have been made to the code, the environment is held perfectly constant, and $P$ is correct for $T$. Algorithms based on such a bias would always select an empty set of test cases for reexecution. Cost-effective safe RTS techniques must utilize biases that lie somewhere in between the null-restriction and null-selection biases.

While this ordering has theoretical usefulness, care should be taken in applying it. The bias underlying a technique might rank high in the general-to-specific ordering, but application of the technique might be impractical. Or, a bias might apply to a wide class of programs, but it might fail to apply to exactly those programs in which we are interested. Nonetheless, the general-to-specific ordering does supply one tool for comparing safe RTS techniques.
3.2 State, Semantic Words, and Agent Sets

So far, we have focused on the definition of safe RTS, rather than the construction of proofs of safety. No proof can proceed, however, without a model of the program and computing environment under test. Because no one model is ideal for all techniques, we present the primitive entities out of which a model and proof specific to a technique can be built. Later in this paper, we present two general models based on these primitives — one based on finite automata, the other on a partitioning process over the intersection of program graphs. These two models are intended to be general, and thus applicable to many different specific models of computation; the primitives we present now, however, are applicable under various models.

Because the framework developed in this section characterizes safe RTS techniques in general, and because analyses of safety involve diverse factors including peripheral devices and other external agents in the computing environment, the following definitions are quite abstract. As we shall show later, to utilize these definitions in assessing a specific safe RTS technique, it is necessary to precisely tailor the definitions in terms of the technique's computational model: such precision is critical for exacting a specification of the technique's regression bias.

We begin by defining “state”:

**Definition 10. State** S: The complete specification of the internal parameters of the computational model adopted for the program and computing environment in question at time $t$.

Normally when we think of “state” we think of the contents of memory and registers. To completely characterize state for our purposes, however, we might also need to include the interfaces between the computer and external universe. In essence, the state should subsume all significant factors internal to the model of the computer, where significance depends on the assumptions of the regression bias. For instance, most testers assume that the transient voltages in gates during a state transition are insignificant and only the stable value matters; there might, however, be an RTS technique that requires the state to model this transition. Everything not included in the state should be either constant or inconsequential for all runs of test cases. Nonvolatile storage on hard disks, for instance, is generally viewed as external to the computing device. As such, we expect this external object to satisfy the quality of being consistent for each test case, thus allowing us to ignore it. When the hard disk is not sufficiently consistent given the subject domain, we must incorporate the disk into the computational model or rerun all test cases for which this inconsistency is an issue. For instance, a test case consisting of two concurrent distributed transactions might not be able to ignore the state of the hard disk, particularly if the transactions are required to be atomic.

Of course, only part of the state might be pertinent to proving particular test cases fault-behavioral equivalent. In particular, there can exist dynamically changing portions of the state $S$ that can have no effect on the output behavior of a test case according to the regression bias and given the program $P$ under test. We shall thus divide the state $S$ into two domains, $X$ and $S - X$, where $X$ denotes the transient state.

**Definition 11. Transient State** $X$: The subset of the state $S$ that, for a particular test case $t$ run on $P$, can be arbitrarily changed without causing $P$ to exhibit different output behaviors.

---

$^5$Existing safe RTS techniques tend to view the computing device as an interpreter for a particular language. Safe RTS techniques requiring stricter control, however, might require that the tester account for hardware interactions. In fact, some techniques might view the “computing device” as a worldwide computer network, with all of the complexity involved in communication now a fundamental part of the state.
$X$ can be a difficult domain to specify. $X$ usually varies over time; for instance, if the stack and heap are viewed as part of the state, program variables in these regions eventually become dead and join $X$. Nonetheless, if a safe RTS technique can show that the execution of $P'$ on test case $t$ differs from the execution of $P$ on $t$ only in $X$, we know that $P$ and $P'$ are fault-behavioral equivalent on $t$. For example, if the only changes between $P$ and $P'$ consist of changes to variables that are never referenced, then all of the changes that form $P'$ involve $X$. Further, specifying $X$ is not always difficult; in Section 5.3, we construct a new safe RTS technique for which substantial portions of $X$ are easily specified.

Our model must also include dynamic primitives that operate on the static primitives constituting the state; however, the model must account for the fact that the dynamic primitives that we think the code exemplifies might differ from the dynamic primitives that the code combined with the computing environment actually generate. For example, suppose we are testing a system that transparently implements remote procedure calls. As testers, we might expect that a remote procedure call would return in bounded time, particularly if the testing process was designed for a centralized system instead of a distributed one, but in other computing environments this might not be the case (usually long delays would result in an error condition on the remote procedure call, but some systems might not implement this protocol) [12]. Our problem, of course, was that our expectations for the behavior of the dynamic primitive were subtly incorrect.

Further, if our description of the dynamic primitives does not encompass all potential dynamic effects, it might be that even if we had the \textit{correct} dynamic primitive, other unexpected behaviors might occur. For example, if we do not incorporate other processes concurrently executing on the computer into the state $S$, and if real-time constraints exist, a round-robin scheduled system cannot be sufficiently specified. An RTS technique based on this insufficient foundation might indicate that a test case need not be re-run when, in fact, running such a test case could reveal a real-time error. Specifications need not only be correct for the domain in question, but they must also encompass all relevant factors. In past work, the roles played by the different components of dynamic primitives have not been explored: this has made proofs more difficult than necessary, as well as hindering the development of a common language for safe RTS techniques.

We thus divide dynamic primitives into three domains: semantic words, valid agents, and rogue agents. An action on the state requires either a rogue agent to act, or a semantic word to bind with a valid or rogue agent.

**Definition 12. Semantic Word:** A semantic word $\sigma$ is the semantic entity representing a set of expected program behaviors, defined in terms of a specific computational model $M$ (which is subject to and an inherent part of the regression bias $R$).

**Definition 13. Agent:** The set of all dynamic effectors that either enact semantic words or enact other behaviors potentially not specified by the program.

**Definition 14. Valid Agent Set:** The set of all agents $\mu$ that transform the expected abstract behaviors of semantic words into real behavior of a physical machine $M$. Valid agents by definition are those that uphold the regression bias $R$.

**Definition 15. Rogue Agent Set:** The set of all agents $\neg\mu$ that either generate actions without binding to a semantic word, or generate inappropriate actions on $M$ given the specifications of $\sigma$. Rogue agents by definition create behavior that is not consistent with the regression bias $R$. 

10
The composition of an agent set with respect to a specific safe RTS technique depends on the way in which the computing environment has been abstracted for that technique. In the case where safe RTS techniques operate on high-level language constructs, the agent set will account for the compiler, the assembler, the underlying hardware and operating system, and potentially other programs executing at the same time. All agents that do not satisfy the expectations of the overarching abstraction assumed by the safe RTS technique pose hazards to the safety of that technique. Also, agents do not necessarily operate at the moment of a program's binding. For example, a compiler transforms a high-level-language statement into a set of assembly instructions, which are then translated into machine language. If this high level language is the level at which a particular safe RTS technique operates, then the compiler itself is a component of the agent set with which a semantic word binds, even though the action of the compiler occurs before the word is executed.

The presentations of existing safe RTS techniques have not included explicit discussions of the nature of the semantic word set or the associated agent sets assumed by the technique, although all have made implicit assumptions about these sets. For instance, as we shall discuss in Section 5, DejaVu and Pythia posit that single statements or blocks of code constitute individual dynamic primitives. As currently designed, these techniques depend on deterministic control-flow-ordered code execution for safety. Of course, safe RTS techniques need not be restricted to the regression biases underlying existing safe RTS techniques. Later, in Section 5.3, we shall discuss a safe RTS technique for spreadsheet languages that does not depend on deterministic, control-flow ordered execution of code, although the outputs of programs in the languages are deterministic. Other techniques might not even require deterministic outputs, as long as certain deterministic constraints on output behavior are met (a random number generator, for example, must produce seemingly non-deterministic output behavior to be correct).

3.3 Iterative Behavioral Equivalence

In general, the problem of proving two programs fault-behavioral equivalent for an arbitrary test case \( t \) is undecidable (trivially Turing reducible to HALT). Thus, safe RTS techniques restrict their attention to a decidable approximation of fault-behavioral equivalence. This approximation must be safe, but it need not be 100% precise.

Given the preceding definitions, we can define the decidable problem of establishing iterative behavioral equivalence; this notion will be crucial to the semantic framework. First, however, we need a definition of an execution trace — one that is more general than the standard control-flow-based definition.

**Definition 16. Execution Trace:** An execution trace records a partially or totally ordered set of semantic words representing the execution of a test case \( t \) on \( P \).

This definition does not mention safety; safe RTS techniques require execution traces that can support inferences about fault-behavioral equivalence. A trace methodology supports such inferences if it makes it possible to correctly infer, from looking at two orderings of semantic words representing the execution of test case \( t \) on \( P \) and \( P' \), that \( P \) and \( P' \) are fault-behavioral equivalent on \( t \). As we discuss later, this requires that the semantic word set satisfy the regression bias for the RTS technique and that the dynamic properties of the programs to which the technique is applied conform to the regression bias.

For most existing safe RTS techniques, an execution trace represents a sequential record of either code statements, basic blocks, or functions executed by a test case, and is assumed to represent a deterministic
execution path through the program. Approximated traces, which are commonly used, need not be in execution order, but if combined with the program $P$ should parameterize a set of execution paths. In general, however, traces for safe RTS techniques need not be restricted to paths. For instance, a safe RTS technique that converts programs into hierarchical plan representations might use execution trees or execution DAGs, where the children of a parent node can occur in any order as long as they all precede the achievement of the parent goal. Whereas two “path” execution traces are identical only when the traces are exact copies, two DAG execution traces are equivalent whenever one of the DAGs can be reordered subject to the partial order and thereby become identical to the DAG that was not reordered.

We can now define iterative behavioral equivalence:

**Definition 17. Iterative Behavioral Equivalence:** Two finite-length execution traces $ET$ and $ET'$ for test case $t$ on programs $P$ and $P'$ are said to be iterative behavioral equivalent if and only if $ET$ and $ET'$ contain equivalent (totally or partially) ordered strings of semantic words.

Iterative-behavioral equivalence imposes stricter requirements for equivalence than does fault-behavioral equivalence. In fact, iterative behavioral equivalence is similar to the all-essential assumption of Leung and White[19]; that assumption asserts that every semantic word in an execution trace can affect the overall path computation. If the abstract behaviors of semantic word set $\Sigma$ actually map correctly by $\mu$ to their corresponding behaviors on real hardware, then iterative behavioral equivalence proves that the piecewise computations of $P$ and $P'$ over test case $t$ remain identical, which in turn guarantees that the final result of the computation is identical.

Note that this definition specifies that execution traces be of finite length, a reasonable assumption given that in practice, any test run must eventually terminate (through either normal program termination or forced termination by the tester). However, some testing processes might create excessively large execution traces, particularly when test cases require long run times: such traces might as well be infinite since we cannot store them. A common solution to this dilemma discards some information from the trace as it is generated — for example, by keeping a bit vector that records whether each semantic word was or was not executed [26]. This culling of information may lead to false judgements of inequivalence, but these judgements can be restricted to conservative judgements that preserve safety.

Clearly, the problem of determining fault-behavioral equivalence is similar to the Post-Correspondence Problem [16] and can be the target of a reduction from the Halting problem. On the other hand, the problem of determining iterative behavioral equivalence is decidable, provided that execution traces are finite: in totally ordered traces, two traces either converge at the next step or they do not, and in partially ordered traces, traces can be reordered to show that one of the equivalent orderings under the partial order converges at each step, or that no such reordering is possible. In fact, we can propose a simple solution to the iterative behavioral equivalence problem based on deterministic finite automata.

To specify this solution, we first consider how to model a technique derived directly from Definition 17. Since iterative behavioral equivalent traces are identical semantic-word for semantic-word (potentially after some reordering if the traces are partial orders), we know that a process of iterating through pairs of words from start to finish will be able to recognize iterative behavioral equivalence. A deterministic finite automaton for safe RTS that directly reflects the definition is inefficient, but subsumes all other techniques for determining iterative behavioral equivalence.
We first create traces for both $P$ and $P'$; this obviates the need for safe RTS since we want to avoid running the test case on $P'$, but in this over-arching method, this redundancy provides a bound on the precision of iterative behavioral equivalence techniques. We then create automata $M$ and $M'$ for the traces $\tau$ on $P$ and $\tau'$ on $P'$, respectively. As we stated previously, these traces are composed of semantic words. We design the automata $M$ corresponding to the trace $\tau$ to accept only the language composed of traces that are equivalent to $\tau$ given the partial order. Thus, in a totally ordered technique, only the trace itself is accepted; in a partially ordered technique, we must create a machine that accepts all traces equivalent under the partial order. As we’ve stated, a trace is equivalent under the partial order if its members can be permuted to create an exact duplicate of the other trace such that no permutation violates the partial order. Such a machine can be exponentially large in the size of the trace, but we are not focusing on efficiency yet: we only wish to show that such a machine is possible.

We then designate $L$ and $L'$ as the languages accepted by $M$ and $M'$, respectively. We know, then, that there is a simple technique for proving equivalence in regular languages: we simply compute the language $L'' = (L \cap \bar{L}) \cup (\bar{L} \cap L')$. Clearly, $M$ accepts the same language as $M'$ only if $L''$ is empty [16]. Obviously, this first approach is not the approach one would choose to implement; nonetheless, all safe RTS techniques based on iterative behavioral equivalence attempt to efficiently approximate the solution produced by this inefficient technique. More significantly, this technique and its second formulation provide the foundation for the efficient generalized safe RTS technique presented in Section 4.

The necessity of obtaining traces for $P'$ is the primary downfall of the first finite automaton formulation. We thus present a second formulation of the finite automaton method, SafeDFA, which, while also not efficient, operates in a manner similar to that of practical safe RTS techniques. First, SafeDFA requires that we convert $P$ and $P'$ into graph representations — we will not be using $P$’s graph, but the trace $\tau$ for $P$ must correspond to an appropriate sequence of nodes under the partial order. Later, in Section 4, we discuss the constraints that these graphs must satisfy to support safe RTS. For the moment, we state only that these graphs must reflect a precedence ordering on the semantic words contained in $P$ and $P'$. This ordering could be total, as it would if the semantic word set and its associated abstractions depended on control-flow order, or it could be partial, as it would if (for example) the ordering reflected only control and data dependencies. The key advantage of this formulation is that SafeDFA can use the graphs to determine (i) whether a test case would definitely create equivalent traces on $P$ and $P'$, or (ii) whether no iterative behavioral equivalence algorithm can determine this given the limitations of the bias, the trace and graph construction functions, and the lack of a trace for $P'$.

First, SafeDFA requires that we transform the trace for $P$ into a finite automaton just as we did for the first finite automaton. For a deterministic path-like trace, this results in a deterministic finite automaton with one state for each semantic word in the trace, where each state has at most one outgoing state transition: the transition from the state to its successor on receiving the successor label. Partially ordered traces require more than one state for each word in the trace. In particular, the automata must include state sequences for every possible sequence of node executions subject to the partial order. Note that this is exactly the same process used to construct $M$ in the first finite automata method.

Next, we create a finite automaton corresponding to the graph representation of $P'$. This graph representation automaton contains a state for every semantic word used in the abstracted program (one state for each occurrence of a semantic word). The automaton built out of these states must contain transitions such
that all traces for $P'$ are accepted by the automaton — remember that both traces and graph are built out of the same semantic word set. The edges in the graph automaton are labelled by their destination state. The automaton might also accept traces that cannot semantically occur; for example, the machine might accept a trace that takes a branch that cannot be taken in the program, perhaps because the condition on the branch is always true. This is perfectly acceptable. The iterative behavioral equivalence process that we will describe will reject any trace that passes through code rendered inert as a result of changes to $P$ to create $P'$: traces that cannot result from the execution of $P'$ but that are accepted by $P''$'s graph automaton must fail to occur as traces for $P$. Thus, we will never compare a graph automaton with a trace for $P$ such that $P''$'s graph automaton accepts the trace, but the trace cannot semantically occur in $P''$.

As a simple example of a graph representation for $P'$, we might create a finite automaton that corresponds to a control flow graph (CFG), except that the edges are labeled by their destinations, rather than by the conditions that lead to that edge being taken, as is the normal convention. The start state of this automaton corresponds to the standard null Start node often added to CFGs. The accepting state corresponds to the null Exit node appended to the end of the CFG. Of course, control-flow graphs are only one example and apply only to one subset of programs. Nonetheless, the core idea holds: the graph formulation must capture the ordered behavior of the program as exemplified by traces.

We now consider the way in which the trace and graph automata are combined to determine whether the test case responsible for the trace must be re-executed. First, consider the simple case in which semantic words in $P$'s trace necessarily imply their successor edge, no matter what other edges exit the node corresponding to the word (i.e., the evaluation of semantic word leads to an edge specified by the value generated by the semantic word, and no two edges with that same condition exist). Given such a constraint, we need only determine whether the language accepted by the trace's DFA is accepted by $P''$'s graph automaton.

In many situations, the requirement that semantic words imply their successor holds. After all, semantic words characterize the behavior of program constructs: in many cases they also characterize the transitions. On the other hand, language designers often give semantic words default successors, which must then be modeled in the graph representation. For instance, in one graphical model of C-style switch statements, a single node might correspond to the switch condition, with edges corresponding to the particular cases. These particular cases are edges where the edge taken is specified by the switch condition. However, if the particular case evaluated by the switch does not exist, then a default edge is taken: this edge corresponds either to a default statement or to a natural exit of the switch. Whether this default edge is taken necessarily depends on what case statements exist in the switch. In a particular program, giving a word a default successor has a well-defined behavior: if the semantic word's alteration to the state does not result in taking one of the labeled edges, take the default edge. When comparing programs, however, default behavior can undermine iterative behavioral equivalence. In particular, suppose that the graph representations for $P$ and $P'$ contain only one difference: one node in $P'$ has an exiting edge that the corresponding node in $P$ does not have. Further, both nodes have default edges. In this case, a trace that was valid for $P$, but which takes the default edge in $P$ in the node corresponding to the changed node in $P'$, can no longer be proven equivalent on $P$ and $P'$ given the techniques presented in this paper. At the same time, however, if the trace does not take the default edge, then the trace is equivalent on those $P$ and $P'$. There are only a small set of possible default situations that can arise, so we can use a simple algorithm and then, if we run into a problematic node, consider the set of possible situations and select on that basis.
While the foregoing automaton does not provide a mathematical upper bound on the precision of iterative behavioral equivalence techniques (that bound being given by the first formulation), it does provide a pragmatic upper bound for decidable safe RTS techniques. The automaton will fail to be as precise as the first automaton only when the threat of default behavior produces imprecision (given that both formulations are dependent on the same regression bias). Since real safe RTS techniques cannot have access to traces for $P'$ either, they too must necessarily accept a certain level of imprecision due to default behaviors.

The validity of the two preceding formulations relies on the existence of a machine $E$ that can tell us when two semantic words in $P$ and $P'$ are equivalent. Of course, since semantic words can themselves be smaller-scale programs, the problem of determining whether two words are equivalent or nonequivalent is undecidable; whenever ambiguity exists, the machine must assume the words are nonequivalent. Fortunately, safety requires only that the machine $E$ tell us the two words are equivalent only if the machine can guarantee that equivalence.

### 3.4 The Stable Semantics Condition and Equivalence Machines

The finite automata methods that we have just discussed prove that a test case can be excluded from regression testing on the basis of its trace given that the semantic words fully determine program behavior. However, semantic words only fully determine actual behavior if all changes to the state result from the binding of semantic words $\sigma$ with valid agents $\mu$. Further, the validity of the finite automata methods depends on the existence of the equivalence machine $E$: safety requires that this machine output an “equivalent” response only when the semantic words always are equivalent when the regression bias is satisfied. The validity of the equivalence machine (and the entire finite automata formulation) is predicated on the premise that the semantic word set satisfies the Stable Semantics Condition, which we will define shortly.

At every point before a semantic word $\sigma$ is executed on the state $S$, a set of properties $p$ holds. These properties describe the state $S$ at that point. By $p$, however, we do not mean exactly the same thing as the state $S$. The state is a static quality of a machine, whereas we use $p$ as a stronger quality that entails that $\{p\} \sigma \land \mu[q]$. Basically, the notation $\{p\} f(S) \{q\}$ describes a specification: if the property $p$ holds and transformation $\sigma \land \mu$ acts on $S$, then $q$ will hold on completion of $f$. We have strengthened this definition somewhat to suggest that $p$ being true implies that $f$ terminates. The set of properties $p$ implies that the action performed by $\sigma \land \mu$ terminates and leaves the machine in a state that satisfies property $q$. Thus, a truth value for $p$ implies that the system performs transformation $\sigma \land \mu$ according to the proper specification.

The fact that property $q$ is satisfied on completion of $\sigma \land \mu$ implies that a subsequent action-pair subject to property $q$ being true can also satisfy a subsequent set of properties on completion, and so on for the length of the program. The set of behaviors of the form $\{p_i\} \sigma_i \land \mu \{p_{i+1}\}$ form an alphabet $\Delta$ of action-pairs. Essentially, this alphabet contains the original alphabet (semantic word set) $\Sigma$ conjoined with a valid regression bias: if the bias holds, then the elements of $\Sigma$ bind with the correct agents, thus generating $\Delta$.

However, when the regression bias does not hold, rogue agents can compromise safety, invalidating finitely automata methods built on the basis of that bias. Safe RTS algorithms such as SafeDFA (and programmers in general) manipulate only the semantic word set $\Sigma$. When these words bind with rogue agents, unsafe situations can occur. As an example, our regression bias might include the assumption that all program statements eventually terminate. On this basis, we might construct a safe RTS algorithm that relies on

---

6 Our notation is derived from Reference [1].
program statements deterministically terminating. Suppose an algorithm based on this bias operates on
the initial and modified versions of a program \( P \) and selects no test cases. But suppose that \( P \) contains a
\textbf{send} statement, and that in the language used for \( P \), \textbf{send} statements block. If we have not forced
the external environment to be rigidly constant when testing \( P \), the receiver of that \textbf{send} statement might never
return a response, and thus the program might perpetually block. This potential nondeterministic behavior
undermines our regression bias and thereby allows fault-revealing test cases to slip past the algorithm. In
terms of our definitions, this occurs because \( \sigma \) bound with \( \neg \mu \), generating a different program behavior than
expected; thus, safety is lost. The key hazard is that the test suite might not, in fact, be DFR for \( P \) if the
environment is not held constant.

In designing the semantic alphabet \( \Sigma \), which models the actions of the program and environment, one
must completely specify the behavior of every construct in the language, including all possible ways in which
the command might interact with environmental factors. If such specifications cannot be made inclusively,
then a safe RTS algorithm must view that word as \textit{always} dangerous.

If a semantic word has a complete and correct specification over the programs and test suites in the bias,
then we say that the semantic word satisfies the \textit{Stable Semantics Condition} (SSC).

Formally, the action-pair \( \sigma \land \mu \), subject to property \( p \), corresponds to a function \( \mathcal{U}(S) \mapsto S \), which causes
\( q \) to be satisfied on completion. The selected function \( \mathcal{U}(S) \) is subject to two constraints:

1. \( \mathcal{U}(S) \) is one-to-one or many-to-one (injective or deterministic, respectively).

2. \textbf{and either} \( \neg \mu = \emptyset \) \textbf{or} \( \neg \mu \) acts only on the transient state \( X \).

We shall say that the satisfaction of these constraints for the semantic words \( \sigma \) implies that the SSC holds
for symbols in the alphabet \( \Sigma \).

An alphabet \( \Sigma \) in which all words satisfy the SSC is necessary but not sufficient for safe RTS. In particular,
the SSC considers behavior only during the action of the word; it does not consider the possibility of rogue
agents generating transformations on the state independently of the executing semantic words. Safe RTS
can occur only when we have an alphabet satisfying the SSC \textit{and} we can show that rogue agents will not
spontaneously transform the state independently of semantic word execution; if the SSC is satisfied, only
valid agents can bind with semantic words, so the rogue agent set is null, provided that rogue agents cannot
spontaneously occur. The equivalence machine we describe next is dependent only on the SSC; the machine
of which it is a part, however, depends on both the SSC and the absence of non-binding rogue agents.

Given a semantic alphabet \( \Sigma \) in which all words satisfy the SSC, an equivalence machine \( E \) must take
a word from each of two execution traces constructed on \( \Sigma \) and determine whether they are equivalent. A
perfect equivalence machine could precisely determine such equivalence, and thus could represent equivalence
by outputting a 1 if the two words are equivalent and a 0 if not. For finite programming languages, it is
possible that such a machine could exist; this would require one to enumerate every possible syntactic
structure and know which structures are provably equivalent to each other. In general, however, and for
infinite languages in particular, no such perfect machine can exist. However, a machine \( E \) can still be made
safe on an infinite language if we weaken the claims it must make. Essentially, we allow \( E \) to output 1 if it
is absolutely sure two words are equivalent, but otherwise output 0. This is safe, as outputting a zero will
merely force tests to be selected that possibly could have been omitted.
To make an equivalence machine's job possible, we must label the symbols in \( \Sigma \) with handles that the machine can recognize. These can be simply the lexical name of a programming language construct if that is the model used, or in the case in which semantic words do not correspond directly to constructs in the programming language, we might provide some other simple handle.

Altogether, this material gives rise to the following theorem.

**Theorem 1.** Let \( \Sigma \) be an alphabet and let \( R \) be a regression bias. If the stable semantics condition holds for \( \Sigma \) and at least some \( \sigma \in \Sigma \) can be deterministically recognized, then an equivalence machine \( E \) can be constructed such that \( E \) never judges two semantic words equivalent incorrectly as long as \( R \) holds.

The proof of the theorem follows immediately from the preceding discussion of the machine's construction.

## 4 A Generalized Safe Regression Algorithm

The primary problem with SafeDFA is that it must create an automaton for the entire trace of each test case and then analyze that entire trace against \( P' \)’s automaton. Traces can be arbitrarily large in the size of the program, so such a technique is inefficient. To generate an efficient technique another approach is needed.

Key to the design of an efficient algorithm is the realization that it is the differences between \( P \) and \( P' \) that cause test case behaviors to diverge. SafeDFA focused on the traces themselves; however, we can deduce the test cases that *might* diverge by first analyzing the program and extracting information that allows us to efficiently select test cases. Many such analyses have worst case costs that are polynomial in the size of the programs. Thus, if we create safe approximations to the traces that are also polynomial in the size of the program, we can generate a technique that is efficient. This is the fundamental goal of the SafeSpace algorithm presented in this section. SafeSpace accepts that a certain level of imprecision is unavoidable if the full traces are not going to be examined, and thus SafeSpace employs a potentially imprecise but efficient approach. Later, we will discuss an extension to SafeSpace that is as precise as SafeDFA and requires polynomial time to analyze the graphs for \( P \) and \( P' \), but requires time polynomial in the length of the full traces to achieve maximally precise test selection.

Figure 2 presents the SafeSpace algorithm. This algorithm constructs abstract graph representations of \( P \) and \( P' \) based on the semantic word set \( \Sigma \), and then traverses these representations in a depth-first manner, constructing a *signature graph* \( G_{\text{sig}} \) that represents the “footprint” of \( P \) on \( P' \): the domain of code over which both programs can be proven iterative behavioral equivalent.\(^7\)

As we shall demonstrate, SafeSpace serves as the basis for the construction of iterative behavioral equivalence algorithms, and acts as a template for the generation of new safe RTS algorithms, by providing a skeleton that can be fleshed out with details specific to a particular safe RTS problem. As such, SafeSpace can be used to abstract behavior and provide safe RTS techniques applicable in program domains that existing techniques have not addressed (declarative visual programs for example). As we shall also demonstrate, SafeSpace can be used as the basis for proofs of safety of existing methods.

\(^7\) This approach is similar to the finite automata based approach used in the safe RTS algorithms developed by Ball [5], but Ball’s approach was developed for control-flow based algorithms whereas our approach is general enough to encompass a wider range of iterative behavior equivalence techniques.
The following sections present details on SafeSpace. The next section defines paired traversals, which underly the algorithm's overall function and correctness. Section 4.2 describes the graph structure used by the algorithm, and the \( \delta \) and \( \epsilon \) functions used to create that structure. Section 4.3 describes details of the graph-walk portion of the algorithm that is encoded in the \texttt{Compare} procedure. Section 4.4 describes extensions to the algorithm that increase its precision.

### 4.1 Paired Traversals

As we have indicated, SafeSpace is based on the idea that no efficient, safe RTS technique can operate directly on traces. Instead, SafeSpace operates on graph representations of programs \( P \) and \( P' \), producing a signature graph of size polynomial in the sizes of the programs. The correctness of this process is predicated on the idea of paired traversals.

A paired traversal is a pair of sequences of nodes \((\tau, \tau')\) created by a simultaneous walk through graphs \( G \) and \( G' \) corresponding to \( P \) and \( P' \). Theorem 2 shows that a paired traversal that has been proven equivalent (i.e., whose constituent sequences have been proven equivalent) inductively up to nodes \( N \) and \( N' \) and then shown equivalent at \( N \) and \( N' \) continues to be equivalent after \( N \) and \( N' \). Note that, similar to our definition
of iterative behavioral equivalence in Section 3.3, paired traversals can contain sequences equivalent under a partial order as well as a total order; in the case of partial orders, however, our inductive proof switches to equivalence not at individual nodes, but to equivalence at the completion of sets equivalent under the partial order. (This is also equivalent to node-by-node equivalence after reordering.) Theorem 2 is significant in that it justifies the subsequent corollary (Corollary 1), which states that all paired traversals proven equivalent up to \( N \) and \( N' \) remain equivalent after \( N \) and \( N' \) if \( N \) and \( N' \) are proven equivalent. This corollary justifies the approximation inherent in SafeSpace: the footprint \( G_{\text{sig}} \) of \( P \) on \( P' \), or the safe space as we shall later define it, acts as a proxy for all traces that execute on semantic words only in its domain.

**Theorem 2.** Given a paired traversal \( (\tau, \tau') \) induced by test case \( t \) on graphs \( G \) and \( G' \), if a node \( N \) in \( G \) and a corresponding node \( N' \) in \( G' \) (as defined by the machines \( M \) and \( E \)) can be shown equivalent under the partial order implicit in the trace for \( t \) by a function \( \iota \) that is equivalent to \( E \) while the stable semantics condition holds, then \( \tau \) and \( \tau' \) will converge to identical states and satisfy identical properties \( q \) after the actions of \( N \) and \( N' \).

**Proof:** We first consider the case of totally ordered traces. If the stable semantics condition holds for all symbols in \( \Sigma \) occurring in the traces, if the rogue agent set is empty, and if the initial set of properties \( p \) can be guaranteed to be equivalent at the start of both traversals, then we know that the transformations on the state engendered by both traces are identical at each step. Thus, the paired traversals defined by the totally ordered traces converge at each step in the trace for both \( P \) and \( P' \).

Partially ordered traces, on the other hand, do not need to converge at each step. However, the definition of the partial order imposes strict requirements that allow iterative behavioral equivalence. In essence, if the traces are equivalent, then one trace can be reordered without violating the partial order, such that we can then perform an equivalence test identical to the total order test and thereby prove equivalence. On the other hand, if the traces are not equivalent, no such permutation exists. As the simulated totally ordered traversal is identical behaviorally to all other permutations allowed, the two traces are in the same family and thus iterative behavioral equivalent. \( \square \)

Note that the reconvergence of partially ordered traces is not the same as the general problem of reconvergence, which we eschewed earlier in Section 3 as undecidable. The general problem allows the violation of the partial order: proving equivalence for the general problem requires a tractable solution to the Post Correspondence Problem.

**Corollary 1.** If \( N \) and \( N' \) can be proven equivalent, then all paired traversals on \( G \) and \( G' \) that are equivalent up to and through \( N \) and \( N' \) are also equivalent after \( N \) and \( N' \).

In essence, this corollary allows us to focus on the graphs and prove equivalence over \( G \) and \( G' \) of sets of traces rather than just a single trace. All traces passing through \( N \) after \( N \) and \( N' \) are proven equivalent remain equivalent over \( G \) and \( G' \) if they were equivalent up to those nodes. The process of establishing paired traversals by simultaneously walking \( G \) and \( G' \) constructs a graph that is isomorphic to subgraphs of
both $G$ and $G'$. We call this graph the signature graph $G_{\text{sig}}$ of $P$ and $P'$, and we call the program space represented by this graph the 	extit{safe space}.

Due to Corollary 1, signature graphs serve as proofs that all test cases whose traces remain entirely within $G_{\text{sig}}$ need not be re-executed. Further, we need not directly "map a test trace onto" this "safe space" to prove that the trace remains entirely within the signature graph. A cutset of edges in $G$ separates the subgraph in $G$ corresponding to the signature graph $G_{\text{sig}}$ from the rest of $G$. We can record this cutset and use it to determine whether a test case needs to be re-executed. If the trace starts in $G_{\text{sig}}$ and does not contain any edge in the cutset, then we know that the test case corresponding to that trace need not be re-executed. This represents a substantial savings in time: the constraints of the problem allow this subgraph isomorphism to be accomplished in low-order polynomial time, and allow the safe approximation of traces by edges only. Further, experiments with existing techniques suggest that in practice this approximation provides results as precise as non-approximated techniques [26].

4.2 Constructing Graph Representations: The $\delta$ and $\epsilon$ functions

The first step of the generalized algorithm 	extbf{SafeSpace} presented in Figure 2 is to transform the program into elements of semantic alphabet $\Sigma$ and then order those elements into a graph structure. These two tasks are performed by the $\delta$ and $\epsilon$ functions, respectively. The $\delta$ function takes the raw program (including code, specifications of environmental behavior, and so on, as discussed in Section 3) and explicitly converts that data into a set of instances of semantic words, as specified by the requirements of the regression bias. The $\epsilon$ function orders the set of semantic words into a graph structure, again according to the requirements of the particular safe RTS technique.

The choice of $\delta$ and $\epsilon$ functions is critical to the efficiency of the algorithm. The $\delta$ and $\epsilon$ functions not only define the size of the graph representations, which in turn affects the execution time and memory space utilized, but they also determine the $\eta$ function used (which implements the equivalence machine $E$), since the $\eta$ function operates on pairs of semantics words just as the equivalence machine $E$ did in 	extbf{SafeDFA}.

The designers of $\delta$ and $\epsilon$ functions must consider whether these functions, as well as the the semantic alphabet $\Sigma$ and regression bias $R$, together suffice for safety. For example, suppose a semantic alphabet designer decided that there would be one semantic word designating all interprocess communications. For some applications this might suffice; other applications, however, might use both blocking and non-blocking send/receive protocols. Clearly, changing the protocol can result in different behavior, which is not considered by this semantic alphabet. Similarly, a designer must ensure that the function $\epsilon$ actually represents all precedence orderings in which one is interested. For example, as we discussed earlier, a partial order might not be sufficiently strong. In the end, the $\delta$ and $\epsilon$ functions must ensure that the Stable Semantics Condition (SSC) holds for the chosen alphabet for all programs covered by the regression bias, or ensure that if the SSC can fail, failure is detectable.

As an example, consider the simple branching construct illustrated in Figure 3. Figure 3-a contains a simple C-style if/else statement; the graph in 3-b illustrates one possible representation of this code. In this case, the $\delta$ and $\epsilon$ functions create a single decision node with three branch edges leading to successor nodes. The 	exttt{if, else if} and 	exttt{else} components of the code are absorbed into one node in this representation. For large and dense if/else statements, such a representation may reflect the branch tables generated in assembly code for the statements [3]. For sparse branch statements, the representation may not reflect the code
1. if (a == 1)
2.    c = 1;
3. else if (a == 2)
4.    c = 3;
5. else
6.    c = 5;

Figure 3: Two graph constructions for a C if/else statement.

generated; Figure 3-c presents a representation reflecting the cascading branch code that might be generated for sparse branch statements.

For many safe RTS techniques, the particular choice of representation in this case is irrelevant; for others, however, the choice may be significant. Suppose, for example, that we are modeling a real-time system in which the compiler constructs a branch table. Perhaps to maintain response time we required the speed of the branch table on large and dense switch statements. The semantic word set used in the graph structure inherently accounts for maximum response times given inputs, but it assigns a particularly low value for the decision node as it assumes an efficient branch representation. If our compiler, actually produces a cascading branch sequence, our specification for the if/else statement is incorrect, and safety may be violated. Of course, for non-real-time systems, the interpretation of the large if/else may be irrelevant. Whatever choice we make is informed by our subject domain and transforms our regression bias.

In Section 5, we consider several safe RTS algorithms, that use various δ and ε functions, and we specify the conditions under which each can operate safely.

4.3 Finding the Safe Space

We next consider the graph-walk portion of the algorithm of Figure 2, encoded in the Compare procedure. This aspect of the algorithm is derived from standard depth-first search algorithms, thus making it structurally similar to DejaVu and Ball's algorithms [5, 26].

The algorithm acts by extending the safe space $G_{\text{sig}}$. When SafeSpace cannot prove that a pair of edges in $G$ and $G'$ lead to a pair of nodes $(N, N')$ that are equivalent under $\iota$, the algorithm changes the destination of the edge to a "Select" node. Any test case whose trace for $P$ cannot be mapped onto $G_{\text{sig}}$ without encountering the Select node must be re-executed. Otherwise, SafeSpace has proven that as long as the regression bias holds, the test case cannot be fault revealing.

Note that the algorithm also adds an edge from a node $N$ to the Select node if the number of edges outgoing from $N$ changes. As a simple example, if we used actual code statements as our semantic word alphabet and control-flow as the $\epsilon$ function, adding a new edge outgoing from a node representing a switch statement (i.e. adding a new switch case) might cause a test case that formerly exercised the default edge to exercise the new edge. This can occur even if the execution trace for $P$ includes no changed entities.

Also, note the use of the "$N'$-visited" condition. The SafeSpace algorithm does not first descend to a pair of nodes and then examine them for equivalence. Instead, it sits at a pair of nodes already proven equivalent and looks across edges at pairs of successor nodes $V$ and $V'$; there, it first checks whether the
successor nodes are equivalent (line 19), and if they are and if \( V \) is marked \( V' \)-visited, it does not proceed further in its traversal beneath \( V \) (lines 21-22). Note that if \( V \) is marked \( X' \)-visited where \( X' \neq V' \), then the trace is looping back on itself for the \( \tau \) trace, but is not looping identically back on itself for the \( \tau' \) trace. We call this the reentrant trace problem and discuss it further in Section 4.4. An instantiation of the SafeSpace algorithm must make a choice for a Reentrant Trace Solution function that acts in this case, and this choice will affect the precision and efficiency of the resulting RTS technique.

How one copes with reentrant traces depends on the instantiation of the SafeSpace algorithm. Safe RTS algorithms in practice have dealt with this problem in many different manners. The Rothermel/Harrold algorithm, DejaVu [26], for instance, continues until the safe space is reentered by both traces at equivalent nodes, or until the technique locates nonequivalent semantic words. In the latter case, DejaVu (effectively) directs an edge to the Select node, even though other test cases whose traces are not reentrant might be associated with that edge. Ball’s family of safe RTS algorithms, on the other hand, address the reentry problem in a sequence of progressively more precise ways that require progressively greater computational expense. Essentially, these algorithms attempt to store extra information about the reentrant trace and thereby improve selection on traces [5]. The basic formulation of SafeSpace presented here always selects test cases on reentering, thus we need not worry about reentrant traces — \( G_{sg} \) is sufficient for selection. Later, we will consider a different choice for Reentrant Trace.

Note that it is the re-entry problem that makes the basic formulation of the SafeSpace algorithm less powerful than SafeDFA. The problem is that if we harvest test cases upon reentering the safe space at different nodes, we might select them for re-execution unnecessarily if it happened that the nodes where they reentered still guaranteed iterative behavioral equivalence. It is possible that if we tracked their mutual behavior further, traces might eventually reconverge to a pair of nodes that are equivalent to the same node in \( G_{sg} \) in such a way that they possessed equivalent states and satisfied identical properties. Then, they would resume execution as one strand that need not be selected for retest. Since the reentrant trace scenario has not been observed to occur in practice [26], we have chosen to select upon reentry for our basic formulation. However, because some techniques might benefit from different solutions, and because proof algorithms often require different solutions, we have incorporated the possibility of extension.

After the recursive calls to Compare initiated at line 7 of SafeSpace complete, the algorithm returns \( G_{sg} \). Previous safe graph-walk algorithms never explicitly constructed \( G_{sg} \), instead they returned edges that attempt to approximate it, or directly selected tests on the basis of those edges. As shown by Ball for CFGs, this form of edge selection must result in imprecision in the worst case [5]. Constructing \( G_{sg} \) and returning it eliminates this particular form of edge-selection imprecision at the cost of storing a copy of \( G_{sg} \); as mentioned above, however, the safe space solution can add a different source of imprecision. Below, we provide a solution that does not suffer imprecision (although after SafeSpace returns \( G_{sg} \) in that solution, selecting a maximally precise test suite requires examining the unapproximated trace, which can be arbitrarily long in the length of the program.

A brief analysis shows that the graph-walk portion of SafeSpace with the basic Reentrant Trace solution operates in \( O(|N| + |E|) \) time, since for identical programs it must check every node and every edge. This provides, then, a basis for gauging the time that should be spent on \( \delta \) and \( \epsilon \) functions in the early stages of the algorithm, since their construction can determine the number of nodes and edges or determine the total running time if the two functions dwarf the algorithm’s run time.
4.4 Proving Reentrant Convergent Traces Identical

The SafeSpace algorithm just described is not as precise as SafeDFA. SafeSpace will select test cases whose execution path reenters nodes in G corresponding to safe space nodes in \( G_{\text{sig}} \) but not in \( G' \), even if the sequence of semantic words executed remains the same. As a concrete example, consider graphs \( G \) and \( G' \) shown in Figure 4. (Start and Exit nodes are omitted from the graphs for simplicity). In these graphs, identical node labels imply that the semantic words represented by the nodes are identical. By following all possible traces through \( G \) and \( G' \) one can see that the programs \( P \) and \( P' \) represented by the two graphs are semantically equivalent. Nonetheless, SafeSpace would prune any test case that takes edge (2,1) in \( G \), because that edge reenters the safe space, while the corresponding edge for \( G' \) takes edge (2',3'), thus continuing on to a node that has not yet been visited (node 3').

However, the semantic word represented by node 3' is identical to the word represented by the node reached on reentering the safe space (node 1), so up until this point, no divergence in the computation has occurred. In fact, for these two graphs, no divergence can occur. We define the problem of determining whether semantically identical traces executing divergent paths remain equivalent and reconverge as the reentrant trace problem.\(^8\)

Extending SafeSpace to provide greater precision in the presence of reentrant traces is arguably impractical. Any extension to SafeSpace will necessarily create a longer run time as the current method always stops on reentry; further, new methods can generate new forms of imprecision. At the same time, no practical cases have been observed in which the reentrant trace problem occurs [24]: all cases examined thus far have been contrived. Nonetheless, it is possible that cases will be found to exist in practice, and that in some situations, the cost of test cases may be sufficiently high to justify even a small possibility of eliminating one. Further, abstractions other than control flow might result in graphs where reentrant traces are more common, and in that case, employing a different strategy might be useful.

Moreover, in proofs, extensions hold a more central location. As we shall discuss in Section 5, an instantiation of SafeSpace used in a proof must have a regression bias that is a subset of the regression bias of the technique being proven safe. If, for example, the technique to be proven safe does not always select on reentry, then the basic formulation of SafeSpace selects some test cases not selected by the technique under

---

\(^8\)Reentrant traces are the factor underlying the “pathological” example of a program in which DejaVu is imprecise, given in [25], although a trace could be reentrant and not result in such a case.
consideration. But this means that SafeSpace might accidentally possess a larger bias than the technique under consideration, and thus be proven safe on a set of programs over which the subject technique is not necessarily safe. In essence, invisible assumptions might slip into the regression bias for SafeSpace, undermining the entire proof.

There are many possible alternatives for the Reentrant_Trace function. Some alternatives might result in inefficient SafeSpace algorithms. In practice, the choice of an extension is based on economic considerations; in constructing proofs, efficiency is irrelevant, and instead the choice should make SafeSpace’s regression bias not only a subset of the bias of the technique being proven safe, but as close a subset as possible.

In this paper, we present only one extension to the SafeSpace algorithm: the Respawning Safe Space (RSS) Solution. This extension makes the SafeSpace algorithm as precise as the SafeDFA method, while requiring only $O(|N|^3)$ time to construct $G_{adj}$. However, the test-selection portion of the extension requires that entire traces be examined. The algorithm also requires a factor of $|P|$ more space than SafeSpace in the worst case, although examples that result in such behavior are rather contrived. We have chosen to present this extension because it is potentially useful both in new techniques and in proofs; in particular, we use the RSS solution in the proof for DejaVu in Section 5.

4.4.1 The Respawning Safe Space Solution

The Respawning Safe Space (RSS) Solution provides an alternative to the original Reentrant_Trace solution presented in the previous sections. The original Reentrant_Trace solution selects an edge when it reenters the safe space; in contrast, the RSS solution continues, “spawning” duplicates of nodes in $G$.

Below, we present the steps that must be taken when the RSS version of Reentrant_Trace is called. In the following, $V$ is a node in $G$ previously marked “$X'$-visited” for each $X' \in \{X_1', X_2', \ldots, X_k'\}$, and $V'$ ($V' \notin \{X_1', X_2', \ldots, X_k'\}$) is the node in $G'$ to which $V$ is being compared. $Y$ is the source node from which Compare was called. The steps in the algorithm are as follows:

1. Duplicate $V$ creating $V_{dup}$ and add a pointer from $V_{dup}$ to the base node $V$. The edge $(Y, V')$ is connected to $V_{dup}$.
2. Mark $V_{dup}$ “originally $V'$-visited.”
3. Add a pointer from $V$ to $V_{dup}$.
4. Mark $V_{dup}$ “$X'$-visited” for all $X' \in \{X_1', X_2', \ldots, X_k'\}$.
5. Following pointers, mark $V$ and all nodes previously spawned off of $V$ “$V'$-visited”.
6. Call Compare on all node pairs reachable from edges exiting $V$, using $V_{dup}$ as the source of these edges (nodes and edges lower in the stack are left as they are).

Figure 5 lets us illustrate this procedure. Proceeding from the Start nodes (omitted for simplicity), the algorithm reaches nodes 1 and 1', finds them equivalent, and adds node 1 to $G_{adj}$, marking it “originally 1'-visited”. Suppose the algorithm next considers the self-looping edges from nodes 1 and 1'; it notes that the labels of the destination nodes (1 and 1') are identical, and that 1 is already 1'-visited, and thus does not walk forward to these nodes. Next the algorithm considers edges (1, 2) and (1', 2'), finds 2 and 2' equivalent,
and adds node 2 to $G_{sig}$ marked “originally 2'-visited”. Only one edge leaves node 2 in $G$ (2' in $G'$); however, in $G$ this edge leads to node 1 whereas in $G'$ it leads to a new node, 3’. In the basic formulation of SafeSpace, the algorithm would note that 1 is already in the safe space and would direct edge (2,1) to Select. With the RSS solution, however, the algorithm duplicates node 1, creating $1_{dup}$ on the “1 stack.” The new node is marked “originally 3'-visited.” Both edges leaving this node (or rather, their corresponding edges in $G$ and $G'$) lead to equivalent nodes in $G$ and $G'$, causing those edges to point to the nodes marked “originally 1'-visited” and “originally 2'-visited”, respectively.9 The algorithm terminates having selected no edges. Note that the edges in the above graphs are not the pointers inserted in the RSS solution. The pointers have not been included in this representation, and exist only to support the node stack data structure.

Notice the different roles played by $P$ and $P'$ in this extension to SafeSpace. $P$ guides the creation of the bottom layer of nodes in $G_{sig}$, corresponding exactly to the concept of the safe space in the basic formulation of the SafeSpace algorithm. However, now $P'$ plays a more active role; not only does $P'$ determine equivalence, it also guides the stacking process in which nodes are respawned off of each other. In addition, $P'$ also determines to which node in the stack a new edge is added. This interaction of $P$ and $P'$ changes the concept of the safe space for this extension to the SafeSpace algorithm. In essence, Respawning Safe Space adds a dimension to the safe space generated by the basic algorithm. Because of this, we call $G_{sig}$ the cross safe space in the RSS extension.

We can bound the run time of this solution, as we create a new copy of a node only when we reenter the safe space such that the node $V$ already in the safe space is not marked $V'$ visited. But we know that at each step, SafeSpace marks one node visited by another node. If we presume that the graph for $P$ contains $n$ nodes and the graph for $P'$ contains $m$ nodes, a node can only acquire $|m|$ visited flags (and that would require a rather contrived program to generate). Thus, at most $|mn|$ steps need to be taken. Further, since each step creates at most $m$ new pointer links between nodes in the stack, with one new edge added from the preceding node to the new node, the total run time is $O(nm^2)$.

Under the RSS solution, once the analysis phase is complete, we select traces just as we did for the regular SafeSpace algorithm: we run a traversal of the cross safe space $G_{sig}$ until the trace reaches the exit node.

---

9This example illustrates a case in which $G_{sig}$ is the same size as $G'$. Clearly, $G_{sig}$ can also be the same size as $G$; to prove this, invert the graphs so that $P'$ is $P$ and vice versa. At this point, one might imagine that $G_{sig}$ is restricted to the maximum of $G$ and $G'$; as it happens, this is not the case. Nodes in $G'$ can appear on different stacks in $G_{sig}$, thus ensuring that the size of $G_{sig}$ can exceed both $G$ and $G'$.
or the Select node. Test cases whose traces end at the Select node are rerun; the rest need not be rerun. Of course, if we want maximal precision, we need to run the entire trace through $G_{\text{sig}}$, which would require time unbounded in the size of the program. Instead, as we suggested for the basic formulation, we can extract information from $G_{\text{sig}}$ and use that to select test cases. In so doing, a certain amount of precision is necessarily lost. The RSS solution, however, is primarily intended for proofs: approximation is unnecessary in many cases since we do not care how much time the algorithm takes — only that its bias is contained in the bias of the technique being proven safe (as we shall elaborate shortly).

## 5 Applications of the Framework

We have presented a framework for safe RTS techniques, including a semantic model and a generalized safe RTS algorithm; next we show how this framework can be applied to real-world problems. In particular, the use of the framework facilitates three tasks:

1. Proving that an existing RTS technique is safe.
2. Defining the set of programs over which an RTS technique is safe.
3. Generating new safe RTS techniques.

First, we provide an example of the application of the framework to the first two problems. We describe and demonstrate a five-step process by which our framework can be used to assess the safety and regression bias of RTS techniques, using DejaVu as a subject. In doing this, we also define the regression bias of DejaVu, thus describing the set of programs over which DejaVu is safe.

Second, to further illustrate the generality of the framework, in Section 5.2 we sketch its application to two additional safe RTS techniques, Pythia and TestTube.

Finally, in Section 5.3, we use our framework to create a new safe RTS algorithm targeting spreadsheet-like languages. Subsequently, we discuss how the techniques of this paper can facilitate the development of safe RTS techniques in other ways: by specifying regression biases, testers should be able to easily recognize safe RTS problems that are close to problems faced in the past, thus allowing reuse of pre-existing techniques.

### 5.1 An Analysis of the Safety and Regression Bias of DejaVu

We now turn to our illustration of the use of our framework for the analysis of an existing RTS technique, DejaVu. Before proceeding with this illustration we briefly summarize the DejaVu technique. Readers interested in greater detail on DejaVu should consult [22, 26].

#### 5.1.1 The DejaVu RTS technique

The DejaVu technique is actually a family of related algorithms for performing safe RTS. One member of this family has been implemented as a tool that has subsequently been evaluated in several empirical studies to determine its precision and efficiency over a variety of sample programs and test suites [7, 21, 26].

DejaVu requires that we first transform the source code for $P$ and $P'$ into control-flow graph (CFG) representations $G$ and $G'$ [3]. The CFGs required by DejaVu use one node for each simple statement in a program, with transitions that model explicit and implicit control flow between statements. Start and
Exit nodes are added to the CFGs to simplify analysis. Depending on the version of the technique used, these CFGs can span either single procedures or multiple procedures. In formulations of DejaVu using single-procedure CFGs, the union of the test cases selected for each procedure constitutes \( T' \).

DejaVu also requires an approximation of a control-flow-based execution trace for each test case \( t \) on \( P \).

**Definition 18. Control-Flow-Based Execution Trace:** A record of the code statements executed by test case \( t \) on program \( P \) in order of execution.

DejaVu does not actually maintain the entire execution trace for each test case; instead, the tool maintains a bit vector that denotes which edges in the CFG \( G \) have been traversed, omitting order of traversal and traversal frequency. Given this trace information, DejaVu executes on pairs of CFGs from identically named procedures in \( P \) and \( P' \), generating a set of dangerous edges that can be used for test selection.

Before describing this process further, however, we define several concepts underlying DejaVu.

**Definition 19. Modification-Traversing Test Case:** A test case \( t \) is modification-traversing for \( P \) and \( P' \) if the execution trace for \( t \) on \( P \) is not identical to the execution trace of \( t \) on \( P' \), given a token by token comparison of the semantic words associated with the nodes in the traces.

**Definition 20. Modification-Revealing Test Case:** A test case \( t \) is modification-revealing for \( P \) and \( P' \) if the execution of \( t \) on \( P' \) produces different output behavior from that produced by \( t \) on \( P \).

**Definition 21. Fault-Revealing Test Case:** A test case \( t \) is fault revealing for \( P' \) if the execution of \( t \) on \( P' \) reveals a fault in \( P' \).

**Definition 22. Dangerous Edge:** A dangerous edge is an edge in the CFG for \( P \) such that DejaVu cannot guarantee that any test case traversing that edge is non-modification-traversing for \( P \) and \( P' \).

As we shall discuss in some detail later when examining DejaVu’s regression bias, by selecting all tests through dangerous edges, DejaVu selects all test cases that are modification-traversing for \( P \) and \( P' \) [22]. As Reference [25] shows, when certain conditions hold, these test cases include all test cases that are modification-revealing for \( P \) and \( P' \), and thereby contain all non-obsolete test cases that are fault-revealing for \( P' \).

Figure 6 presents one of the more basic DejaVu algorithms presented in [22]. Numerous optimizations have been added to this algorithm to make DejaVu more precise, as discussed in [22, 26]. For our purposes, however, this algorithm suffices. The algorithm operates by simultaneously traversing CFGs \( G \) and \( G' \). If the traversal of an edge leads to a pair of nodes \( < V, V' > \) such that the statements represented by those nodes differ, then the algorithm adds the edge leading to that pair of nodes to the dangerous edge set. In addition, if the algorithm reaches an edge in \( P \) that does not have a corresponding edge in \( P' \), it places that edge in the dangerous edge set. Further, for this simple version, if the node \( V' \) in a node-pair has an outgoing edge that does not match an outgoing edge of \( V \), this version of the algorithm adds all edges leading to \( < V, V' > \) to the dangerous edge set. Such extra edges in \( P' \) must result in the preceding edges being selected, because such extra edges might represent changes in behavior due to additions of new switch cases.
algorithm SelectTests($P, P', T); T' 
input $P, P'$: base and modified versions of a procedure 
output $T$: the set of dangerous edges 
1. begin 
  2. $T = \phi$ 
  3. construct $G$ and $G'$, CFGs for $P$ and $P'$, with entry nodes $E$ and $E'$ 
  4. $\text{Compare}(E, E')$ 
  5. return $T$ 
  6. end 

procedure $\text{Compare}(N, N')$ 
input $N$ and $N'$: nodes in $G$ and $G'$ 
7. begin 
  8. mark $N$ “$N'$-visited” 
  9. for each successor $C$ of $N$ in $G$ do 
  10. $L =$ the label on edge $(N, C)$ or $\epsilon$ if $(N, C)$ is unlabeled 
  11. $C'$=the node in $G'$ such that $(N', C')$ has label $L$ or null if no such edge 
  12. if $C'$=null 
  13. $T = T \cup (N, C)$ 
  14. else if $C'$ is not marked “$C'$-visited” 
  15. if $\text{Equivalent}(C, C')$ 
  16. $T = T \cup (N, C)$ 
  17. else 
  18. $\text{Compare}(C, C')$ 
  19. endif 
  20. endfor 
  21. for each edge $(N', C')$ lacking a corresponding edge $(N, C)$ do 
  22. locate all directed edges $(X, N')$, where $X$ is any vertex 
  23. $T = T \cup (X, N)$ 
  24. endfor 
  25. endif 
  26. end 

Figure 6: DejaVu Algorithm.

5.1.2 Analysis and proof of safety

At this point, we wish to provide an example of a proof of safety involving SafeSpace. Such proofs are predicated on the safety of the general formulation of SafeSpace; given components that are safe — safe $\delta$, $\epsilon$, and $\iota$ functions, for instance — we have shown that the general formulation of SafeSpace is safe (it selects all test cases whose traces exit $G_{exit}$). Given SafeSpace’s safety, we can construct a proof of safety for another technique by generating a proxy algorithm for that technique. This proxy is built using SafeSpace as a structural guideline, but using components (semantic words, state models, $\delta$ functions, etc.) that emulate the behavior of the technique to be proven safe.

Let $A$ be an RTS technique that we wish to prove safe, and let $A_S$ be the target regression bias for $A$. To prove that $A$ is safe, we construct a proxy algorithm $I$ — an instantiation of the SafeSpace algorithm designed to emulate the specified behavior of $A$. This emulation need not be exact: $A$ might make approximations that cause it to select a larger number of test cases than $I$, but as long as the set of test cases selected by $A$ is a superset of the set selected by $I$, then if $I$ is proven safe, we know that $A$ is safe.

To ensure that a particular SafeSpace instantiation $I$ is appropriate for our proof, we must complete two overall tasks:

---

10 The target bias $A_S$ for $A$ need not be the same as the the de facto bias of $A$ – we might be interested in proving the safety of $A$ for only a subset of the programs and testing methodologies to which $A$ in fact applies. However, the target regression bias must be contained within $A$’s actual bias; for the rest of this presentation we make the assumption that this is the case.
1. Prove that $I$ is safe over regression bias $I_B \supseteq A_B$.

2. Prove that every test case selected by $I$ is also selected by $A$.

It might seem that we are substituting one hard problem for another equally hard problem in step 1. However, SafeSpace was deliberately designed to enhance clarity, depending primarily on the conceptually simple idea of a safe space (clarity was, in fact, an important reason for restricting the cross safe space to an extension). Further, we already have proven SafeSpace safe as long as its components fulfill the restrictions discussed in Section 4. Thus, to complete the proof, we need only to construct the component functions and descriptions of the primitives used in the technique, while proving that these pieces of the technique fulfill the requirements of the stable semantics condition and prevent rogue agents.

Of course, in step one, we might discover that we cannot create a technique $I$ safe over $I_B \supseteq A_B$ using components that emulate $A$. This tells us that the basic primitives and logic of $A$ are insufficient for safety.

We describe a five-step proof process for RTS techniques based on the logic described above, but in this description we emphasize the iterative nature of this process. In particular, the logic described above demands that we have a definition of technique $A$'s regression bias, that is, the programs and test suites over which $A$ must be safe. In practice, however, we might begin the proof process not fully understanding what characteristics actually reflect this bias, or as the process proceeds, we might find that our initial assumptions about this bias were either insufficient or overly general. A fundamental part of the safe RTS process is the establishment of a regression bias, while a fundamental part of a safe RTS proof is establishment of safety over a well-defined bias. Thus, at every step of the proof, we must consider whether the bias is sufficient for the task at hand.

**Five-step proof process:**

1. Establish the regression bias $A_B$ for which RTS technique $A$ is to be proven safe.

2. Specify a state model $S$ and a semantic word set $\Sigma$ that fulfill the regression bias. Explicate the nature of the agent sets $\mu$ and $\neg \mu$ in relation to this bias (consider ways in which the semantic word set and state model might fail to reflect real program behavior).

3. Construct safety-preserving $\delta$, $\epsilon$, and $\iota$ functions out of $S$ and $\Sigma$.

4. Transform trace information into abstract execution traces for SafeSpace using the semantic word set $\Sigma$ and the ordering implied by the $\epsilon$ function.

5. Show that the SafeSpace instantiation $I$ established in steps 2 through 4 has a regression bias containing that of $A$, and show that if $I$ selects a test case $t$, then $A$ selects $t$.

Note that the SafeSpace instantiation of the safe RTS algorithm $A$ need not use the same state model or semantic word set used by $A$. In general, it may be easier to prove a technique safe if the general algorithm views the code in the same way, but that need not always be the case. In particular, if the structure of an RTS technique differs from depth-first search, the primitives must be adapted so that depth-first search makes sense. Alternatively, a different formulation of the generalized safe RTS algorithm could be used.
Step 1: Establish DejaVu’s Regression Bias

DejaVu imposes several requirements on subject programs. First, DejaVu requires that all test cases that are fault-revealing for \( P' \) be modification revealing for \( P \) and \( P' \), and that all test cases that are modification-revealing for \( P \) and \( P' \) be modification traversing for \( P \) and \( P' \). As noted in Section 5.1.1, it has been shown that under certain conditions, DejaVu selects every test case in \( T \) that is modification-traversing for \( P \) and \( P' \) [22]. Thus, as a first approximation to its regression bias, we can say that DejaVu is safe in all situations in which all fault-revealing test cases are modification traversing.\(^{11}\) However, we still need to specify the types of programs for which this is true and the types for which this is not true.

In general, fault-revealing test cases are not always modification-traversing. In order for the fact that a test case \( t \) is fault-revealing for \( P' \) to imply that \( t \) is modification-traversing for \( P \) and \( P' \), \( P \) must be correct for \( T \).\(^{12}\) Otherwise, \( P \) and \( P' \) might exhibit equivalent behavior (and traces) on \( t \), but \( P' \) might exhibit a fault on \( t \), inherited from \( P \). DejaVu’s regression bias includes the assumption that faults occur deterministically; thus, if test suite \( T \) was run on \( P \) and all faults discovered in \( P \) were corrected without introducing new faults revealable by \( T \), \( P \) is correct for \( T \).

Two problems arise as a result of this bias: first, an attempt to correct a fault might be made, but the attempt might fail; second, nondeterministic behavior might hide a fault still remaining in the system. The first problem is easily rectified: test cases that exposed faults in \( P \) can be rerun for \( P' \), regardless of the results of DejaVu’s analysis. Nondeterministic test cases (see e.g. [10]), however, are not so easily handled by DejaVu. Nondeterministic test cases explore different input-output relations on different runs. In particular, on different runs, such test cases might explore different regions of the input space or result in the execution of different paths through the code. When such nondeterminism exists for a test case, we cannot assert that correct execution of the test case proves that the test case cannot exhibit a fault.\(^{13}\)

One dichotomy of nondeterminism is of particular interest: explicit nondeterminism versus implicit nondeterminism. Explicit nondeterminism occurs when software designers intend code to behave nondeterministically and can control this behavior within the testing process. Implicit nondeterminism occurs when the software is sensitive to aspects of the environment that cannot be strictly controlled in testing environments.

As a simple example of explicit nondeterminism, random-number generators create nondeterministic sequences of numbers. The key feature of explicit nondeterminism is that testers can “freeze” the behavior of the nondeterministic segment of code. In general, explicit nondeterminism can be dealt with by calculating one valid deterministic behavior and replacing the nondeterministic code with code that reproduces that effect. For instance, a pseudo-random-number generator can be presented with the same seed on every run of the program.

Implicit nondeterminism, on the other hand, represents the worst of the rogue agents \(-\mu \). Implicit nondeterminism includes any type of random behavior that cannot be localized, predicted, and controlled. For instance, in a real-time program, page faults present possibly significant effects on response time, but usually

---

\(^{11}\)Note that we begin our analysis of DejaVu with a great deal of knowledge about DejaVu’s safety; proofs for new techniques, however, might begin with significantly less knowledge.

\(^{12}\)Also, as we discussed in Section 2, obsolete test cases must be removed from \( T \). Obsolete test cases will almost certainly be modification-traversing, but need not be fault-revealing. Obsolete test cases should either be disposed of or incorporated into the process of testing new functionality — that task, however, is not the subject of this paper.

\(^{13}\)Strictly speaking, this is not absolutely true. There might be cases where it is known that only a small set of nondeterministic behaviors can occur; if we see the entire set over multiple runs then we know that each possibility has been exercised. In general, however, nondeterministic test cases are not so well-behaved.
cannot be predicted. Similarly, in multi-threaded programs, statements may be executed in nondeterministic order, resulting in nondeterministic behavior when shared variables are incorrectly accessed or when output behaviors are not properly synchronized.

Unlike explicit nondeterminism, implicit nondeterminism forces us to refine our regression bias. Our initial definition of DejaVu’s bias required only that a test case being fault-revealing must imply that it is modification-revealing, which in turn must imply that it is modification-traversing. Satisfaction of this requirement requires that \( P \) be correct for \( T \). However, we know that for \( P \) to be proven correct for \( T \), \( T \) must be deterministic for \( P \). Nondeterminism is not consistent with this requirement. Thus, DejaVu’s regression bias includes only those programs that can be transformed by trivial changes (where trivial depends on constraints on the testing process) into programs that behave deterministically in all significant ways. Note that this bias restricts not only the code constructs used, but the type of the program as well. For instance, applying DejaVu to an arbitrary real-time program seems difficult to accomplish while preserving safety. We cannot trivially change a real-time program to behave deterministically; if we did perform such changes, the changes would violate the timing-dependent nature of the program.

Specifically, DejaVu is obviously safe only for programs that compute values in a time-independent fashion. In other words, DejaVu is not obviously safe for multi-threaded programs, programs that poll the outside environment for nondeterministically arriving input, or any other program that might exhibit different execution traces on different runs due either to varying inputs or varying internal states in the computer. Many programs, however, act in linear fashion, or react consistently to input regardless of when the input occurs. Over these programs, DejaVu’s regression bias holds.

In addition, DejaVu targets only imperative languages. DejaVu uses control flow graphs to select dangerous edges. In other languages such as spreadsheet languages (as we shall see), control flow is not relevant.

DejaVu’s regression bias must be considered in relation not only to programs, but also to testing methodologies. For example, since DejaVu approximates traces with bit vectors, if a test methodology for a program that uses polling could guarantee that each input either consistently arrived before the polling structure asked for it, or consistently arrived after at least one loop through the poll was processed, DejaVu could operate safely on that program. Without this guarantee, a section of code might be executed by some runs of a test case, but not by other runs, thus generating faulty traces. With the above constraint, all traces would be identical since DejaVu records only the statements executed, not their frequency. Testing methodologies are an integral part of the regression bias for a safe RTS technique and constitute part of the space over which a bias is valid. Some software might be amenable to the safe application of DejaVu with one testing methodology but not with another.

**Step 2: Specify the State Model, Semantic Word Set, and Agent Sets**

At this point, we have specified our expectations for the regression bias of DejaVu based on preliminary considerations of nondeterminism. We should now explicitly consider the mapping of the code to primitives that is assumed by DejaVu, in order to expose further potential threats to the safety of the algorithm. Since DejaVu uses fairly simple abstractions of the code, this mapping is fairly simple. Analysis techniques operating on higher-level abstractions such as specifications or interpretations of programs as plans would need to explore the relation of primitives to code and computing environment in greater depth.

First, because DejaVu does not incorporate aspects of the external environment into its semantic word
if (aExpr) statement1; else statement2 (statement1) jmp L2
L1:
L2:
beq $r0, $0, L1
addi $r0, $r0, aExpr

Figure 7: An if/else statement and corresponding MIPS assembler code. The term aExpr represents a value in a register, and strictly speaking actually refers to a particular register.

set, DejaVu must assume that the external environment is constant: therefore, the external environment is not part of the state for DejaVu. Thus, we can use the standard definition of state as program memory. Since DejaVu’s bias does not contain programs that are dependent on high response times, we need not concern ourselves with caches or virtual memory (beyond assuming that they are coherent). Further, we will assume that the computing environment is identical between the execution of P and P’. Specifically, this requirement entails that we use the same compiler, operating system and machine configuration to build and test P’ that was used to build and test P.

In general, the code of P and P’ will be in the subset X of the above state, while live data values will be in S - X. Placing the code in X might seem counterintuitive since the code directly affects program execution. However, the program that we are studying is viewed as external to the state; thus, the code in this case can affect the program only if the program can rewrite its own code, which is generally not the case for languages such as C in modern computer systems.

The semantic word set is even more simple. DejaVu examines code at the statement level; thus, a semantic word corresponds to a single (simple) code statement. There is one semantic word for each normalized code statement that is not lexically identical to another word already in Σ. Thus, there are an infinite number of possible semantic words in Σ, although any particular program contains only a finite number.

The agent sets are associated with the properties that must hold, or that fail to hold for rogue agents. In other words, the valid agent set consists of boolean expressions such as EQUIVALENT_COMPILER ∧ EQUIVALENT_OS ∧ ..., but only when these expressions are true. Rogue agents consist of the same expressions when the expressions are not true. In essence, when a semantic word is bound with an agent, the truth of the boolean expression determines whether the agent is valid or rogue. These expressions also hold between the executions of semantic words — recall the properties p that must hold between semantic word executions. If a rogue agent acts on the non-transient state independently of the execution of semantic words, it violates the model of program behavior.

Consider the specification of a simple set of semantic words in DejaVu: the set of words originating with the if/else construct. In the following, we assume that the set of potential expressions in the if/else have already been proven safe or unsafe. If the expression is not safe, we automatically reject test cases traversing the if/else regardless of the safety of the if/else construct itself. Given the safety of potential expressions, we proceed to prove that if/else is a safe construct with respect to DejaVu. An if/else statement has at least one clear interpretation, as given in Figure 7 [20].

Assuming that this format of the if/else is the only format produced by the compiler, and ignoring the existence of additional else-if clauses for brevity, we can readily specify the behavior of the if/else

\footnote{The Figure uses the MIPS assembly language. Equivalent code could be generated for any modern assembly language.}
construct. The expression aExpr has already been proven safe or unsafe; thus, where aExpr is safe we need only worry about the if/else construct itself. Given then that aExpr is safe, aExpr correctly produces a value (as specified by the peculiarities of the C language). If aExpr is not equal to 0, the beq does nothing and statement 1 is executed; otherwise, the beq changes the program counter and statement 2 is executed. After the execution of statement1, a jump occurs to a label following statement2. No other behaviors are possible if assembly code fully characterizes behavior, as it must if the regression bias for DejaVu holds. Thus, the if/else is deterministic and the if/else construct is safe for DejaVu.

Of course, specification in terms of assembly language is only one approach to proving safety. Formal proof methods could be used for each statement, or informal natural language descriptions could be used, depending on the demands placed upon us by the testing situation.

**Step 3: Construct δ, ε, and ι Functions**

To use an instantiation I of SafeSpace to prove the safety of RTS algorithm A, we must construct the δ, ε and ι functions to be used by I. These functions may duplicate processes used by A, or they may simplify these processes; the key to constructing the functions is that in all cases in which A's regression bias holds, the functions must ensure that I selects a set of test cases that is equivalent to or a subset of the set of test cases selected by A. By ensuring that these functions maintain this relationship, we ensure that a proof that I is safe implies that A is safe.

We begin with the ι function. For the instantiation of SafeSpace that we are here constructing, the ι function arises naturally from the description of the semantic word set. In DejaVu, two semantic words are identical if the normalized lexical values of the words are identical. Thus, in the DejaVu instantiation of SafeSpace, given two semantic words (statements), once we remove comments and spurious white space we simply compare the two statements to determine lexicographic equivalence.

That this choice of ι suffices for safety depends also on the rest of this five-step process, but for the moment, we can partially justify the choice in and of itself. In Section 4, we stated that the ι function must output “equivalent” only when two semantic words produce equivalent behaviors given identical states. Since we either have no nondeterministic constructs in the program, or automatically select any test case encountering them, ι will operate only on deterministic constructs. But since ι outputs “equivalent” only when two statements are lexicographically identical, the behaviors of equivalent words will be identical. Thus, this ι function meets the requirements specified in Section 4. Note that equivalence in this context does not imply that the same transitions are taken out of two nodes when the nodes are equivalent under ι. The conjoinment of the ι and ε functions is needed for that.

Now, turning to the δ and ε functions, these functions act on the code and program environment to create a graphical abstraction of the program. Arbitrary graph representations might not be sufficient for safety over the regression bias in which we are interested. Thus, as we specify δ and ε functions, we must consider how they mesh with SafeSpace as a whole. The δ function must break up the program into nodes that can be processed by the ι function. Similarly, the ε function must order these nodes into a graph such that the intersection process creates a meaningful and safe $G_{sig}$. Note that SafeSpace selects not only on the basis of the ι function, but also when considering edges out of a node: the justifications for selection at this stage must arise naturally out of the ε function and must preserve safety. However, the tasks of δ and ε are not just to preserve safety, but also to construct a SafeSpace instantiation with a simple regression bias that
is contained in the true regression bias of $A$. Preferably, such a bias should be simple enough to transcribe
into a set of guidelines to be used when evaluating whether $A$ can be used safely on a particular program.

The $\delta$ function for our SafeSpace instantiation of DejaVu is rather simple: in fact, we specified it when
we constructed our $\iota$ function. The $\delta$ function maps individual code statements to nodes. Note that the
definition of a statement is somewhat idiosyncratic as used by DejaVu: an entire while loop is not a single
statement, for instance, but rather one statement for the first line of the while, and one statement for each
statement contained in the body of the while loop. This is necessary, however, to allow the use of $\iota$ at a fine
granularity. As we stated earlier, the $\epsilon$ function orders statements, creating links between nodes as specified
by the partial or total order. In the case of the instantiation of SafeSpace that we are constructing for
DejaVu, $\epsilon$ implements a total order that structures the nodes into a control-flow graph. Note that cases
where edges have been added or deleted are already dealt with safely by the SafeSpace algorithm.

We have previously shown that our $\delta$ function suffices for safety. A safe $\epsilon$ function, meanwhile, must link
nodes together such that an unapproximated test trace can be mapped into the signature graph such that the
Select node is never encountered if the test case need not be re-executed. In the case of DejaVu, a control-flow
ordering is sufficient for safety. DejaVu itself, of course, uses approximated traces, but as we stated earlier,
a more precise interpretation of the algorithm is often desirable; we thus decided to create a SafeSpace
instantiation using full traces. In step 5 of the proof process, we will show that the test cases selected
on the basis of full traces are also selected by the approximated traces, although the approximated traces
might result in extra test cases being selected. In Section 5.2.2, we describe a different graph formulation
for TestTube; this same process could have been followed for DejaVu as well, although the safety of DejaVu
might not have been as easily demonstrated by that approach.

Note that, although it turns out that the $\delta$ function is simpler than the $\epsilon$ function for DejaVu, this need
not always be the case. Some abstractions might have very simple orderings, but might require pervasive
global transformations by the $\delta$ function.

Step 4: Construct Abstracted Execution Traces

Although SafeSpace itself does not perform test selection — relegating that task to a subsidiary process
that uses $G_{sig}$ — we must ensure that our execution traces actually suffice for safe test selection on the basis
of our signature graph. Abstract traces are constructed out of the same building blocks as the graphical
representations for $P$ and $P'$. Further, they obey the same precedence rules, except that traces are paths
through the graph rather than graphs themselves. Thus, traces are mapped to a string of semantic words.
The key to the safety of a trace lies in the correctness of the semantic word set and the ordering of the
graph. If the semantic word set satisfies the SSC and no rogue agents transform the state between semantic
words, and if the traces can be embedded in the graph $G_{sig}$ (which means that no edges to Select occur in
the trace), then the test case can be safely excluded.

Given the primitives and the $\delta$ and $\epsilon$ functions for DejaVu, the construction of abstract traces from
real trace data is simple. We convert existing statement-level trace information into the abstract semantic
word set as with $\delta$, ordering the primitives in the trace by their sequence of execution. Note that this
transformation is not necessarily affected by $\delta$ or $\epsilon$. The $\delta$ and $\epsilon$ functions act on $P$ and $P'$, constructing
graphs $G$ and $G'$. The execution traces, on the other hand, are created out of information gathered from
previous executions of $P$. Oftentimes, however, the creation of execution traces is similar to the creation of
$G$, and it involves constructing a subgraph of $G$ using information that corresponds directly to subsets of $P$.

That these traces suffice for the role of safe traces in a SafeSpace instantiation for DejaVu is trivial given that $G_{sib}$ represents a subset of a control-flow graph $G$ (or as it will later turn out, the cross safe space $G_{sib}$ of $G$ and $G'$). A trace will end up in the Select node unless it can make its way through $G_{sib}$ without traversing a Select edge.

**Step 5: Show that ($I_B \supseteq A_B$) and that ($I$ selects $i$) $\Rightarrow$ ($A$ selects $T$).**

The five steps of the proof process guide us towards a regression bias for a technique that is amenable to a SafeSpace instantiation. Once a viable bias $A_B$ has been specified, the SafeSpace instantiation $I$ must be proven safe over that bias. The process of constructing $I$ defines a bias $I_B$ for $I$, and given this bias we must show two things: (5.1) that the bias $I_B$ over which we proved $I$ safe is equivalent to, or a superset of, the desired bias $A_B$ of $A$ (which itself might be a proper subset of $A$’s real bias); and (5.2) that $A$ selects at least the same test cases $I$ selects over $A_B$.

Where (5.1) is concerned, it may not be obvious that $I_B$ can be made an actual superset of $A_B$: this, however, is where we take advantage of the fact that $A_B$ is a specified bias rather than a de facto bias for a real technique. Because a specified bias can strictly detail the properties of the set of programs and test suites that it contains, $I_B$ can be made a superset, even though the bias of the real technique $A$ might contain bizarre outlier programs over which it is accidentally and unjustifiably safe. This is sufficient for our purposes, because we are interested only in proving a technique safe for a particular regression bias: the actual regression bias is purely of academic interest. We still need to ensure that the bias of $I$ is contained in the bias of $A$, but this can be done by ensuring that $I$ is always at least as precise as $A$, as we discuss in step 5.2.

Where (5.2) is concerned, we must show that a fault-revealing test case is selected by $I$ over $A_B$ (or equivalently, given 5.1, over $I_B$) only if $A$ also would select that test case. Given this criterion, $I$ must be at least as precise as $A$ over the programs in $I_B$. If $I$ is more precise than $A$, this means that $A$ is safe over $A_B$ as well. To ensure this containment for DejaVu, we use the RSS extension to SafeSpace presented earlier.

Unlike the basic formulation of SafeSpace, DejaVu selects edges only when it encounters nodes that are lexically different, or encounters nodes with changed numbers of edges or different labels on edges. This is similar to the RSS solution that we discussed in Section 4.4.1, except that DejaVu does not construct a cross safe space. Instead, DejaVu selects all test cases whose traces contain the edge that lead to the changed nodes, even if those particular test cases are not reentrant. To prove DejaVu safe over $A_B$, given the steps already completed, we need only show that on programs and test suites in the space covered by $I_B$, DejaVu selects a test case whenever $I$ selects a test case.

As Figure 2 shows, SafeSpace adds edges that point to the Select node in only four places: lines 13, 17, 20, and 24. Line 13 corresponds to the case when node $N'$ has edges that are not possessed by node $N$ in the original graph $G$. In this case, because of the possibility of default edges, SafeSpace must select the edge leading to that node pair. DejaVu selects on identical edges due to lines 22-24 in the DejaVu algorithm (Figure 6). The second case, line 16, occurs whenever there is an edge $e$ exiting the node $N$ that does not have a counterpart in $N'$, the node corresponding to $N$ in the modified program $P'$. DejaVu selects these edges in lines 12-13. In the third case, line 20 of SafeSpace, an edge $e$ is added if the node $N$ reached in $P$ is not equivalent to the node $N'$ reached in $P'$ via that edge. In the case of this instantiation of the SafeSpace
algorithm, the \( t \) function is lexical equivalence, which means that any edge leading to lexically different nodes is selected. Notice that in lines 15-16 of the DejaVu algorithm, exactly these edges are selected. The final case, line 24 of SafeSpace, does not result in the selection of any test cases in this instantiation since we are using the Respawning Safe Space solution. Note that when DejaVu selects the test cases associated with an edge, it selects all test cases associated with that edge. When SafeSpace using RSS selects the test cases on an edge, it selects only test cases that pass through the node on the stack connected to that edge. Thus, DejaVu selects more test cases than SafeSpace, but it selects all of the test cases selected by SafeSpace.

5.2 Analyses of Other Safe RTS Algorithms

DejaVu is only one example of a safe RTS algorithm. In this section, we briefly consider two other existing algorithms, Pythia[29] and TestTube[11], in the light of our framework and the previous analysis of DejaVu. In doing so, we gain further insights into the use of the framework.

5.2.1 Pythia

First, we shall focus on the safe RTS technique that bears the closest resemblance to DejaVu: Pythia. Our fundamental goal is to briefly show that Pythia's regression bias is essentially the same as DejaVu's; in so doing, we suggest how our framework could be utilized in constructing a proof of safety for Pythia.

Pythia performs safe RTS on C programs. Like DejaVu, Pythia proves equivalence using lexical identity. Instead of generating CFGs and then performing analysis on them, however, Pythia uses commonly existing Unix tools to directly perform textual differencing on the subject programs[29]. Pythia first uses the Unix tool pretty to normalize the source files for \( P \) and \( P' \). Pythia then divides the normalized source code into basic blocks; these basic blocks constitute the semantic word set for a instantiation \( I_P \) of SafeSpace modeling Pythia. The state and agent sets for this instantiation of SafeSpace are identical to those used in DejaVu's instantiation of SafeSpace. Because Pythia uses basic blocks, Pythia's execution traces are identical to DejaVu's except that the elements are basic block transitions rather than statement transitions. Like DejaVu, Pythia relies on a transition model based on control flow between basic blocks to determine sequencing; however, for Pythia, this transition model is not explicitly constructed, but determined implicitly from the syntax of code statements as required during the test selection process.

Given normalized files divided into basic blocks, Pythia executes the Unix tool diff on the two normalized programs \( P \) and \( P' \), attempting to locate blocks that have changed. In the simple case in which the internal structure of blocks has not changed between \( P \) and \( P' \), Pythia provides a solution as accurate as DejaVu. In the case where block structure has changed, Pythia defaults to the only safe alternative given its level of analysis and selects the block(s) that can be determined to safely precede the changed block in terms of control flow. This approach can create greater imprecision than that exhibited by DejaVu.

Given this description and the close relationship between Pythia and DejaVu, it should be apparent how to construct a proof of safety for Pythia. Essentially, a SafeSpace instantiation \( I_P \) for Pythia would be identical to the instantiation \( I \) used above for DejaVu, except that the set of primitives would be composed of basic blocks rather than code statements. However, since a basic block is just a linear set of code statements, the graph for \( I_P \) would simply be a condensed form of the graph used by \( I \). Because we are not interested in efficiency in a proof, \( I_P \) need not select the preceding basic block if block structure changes; that inefficiency is a result of using the Unix tool diff and not traversing blocks in control flow order — this also further
illustrates that a proof of safety of an RTS algorithm $A$ may utilize a SafeSpace instantiation of greater precision than $A$. As a result Pythia's $I_P$ selects the same test cases selected by DejaVu's $I$.

Thus, clearly, Pythia's SafeSpace instantiation $I_P$ is safe over the same regression bias $I_B$ as DejaVu's SafeSpace instantiation $I$. Further, since the instances of Pythia's and DejaVu's imprecisions with respect to their SafeSpace instantiations are unlikely to result in extensions to the types of program covered, the regression biases for DejaVu and Pythia are the same.

5.2.2 TestTube

TestTube performs safe RTS by using lexical analysis at the level of symbols in the global namespace of C programs [11]. For TestTube, only functions, macros, types, and global variables exist. The current version of TestTube relies on the existence of several subsidiary tools, including a program instrumenter (App), a program entity database (CIA), and the set of scripts that constitutes TestTube itself.

Like DejaVu and Pythia, TestTube relies on lexical equivalence, although TestTube's approach is simpler. The CIA database associates checksum information with the entities in the global namespace for both program versions. TestTube compares the checksums of all identical global symbols in the two program versions to determine which ones differ. TestTube then determines which other entities could be influenced by the differences in the changed entities.\(^\text{15}\) Thus, for instance, a global variable used in a changed function is henceforth considered changed itself; functions called by a changed function, however, are not thereby considered changed in and of themselves.

The execution traces used by TestTube are generated by the App instrumenter, which records the set of functions invoked during each test. This information is used in conjunction with the CIA database to generate a list of entities potentially affected by changed functions. This list and the entity difference list generated by CIA for the two program versions indicate whether or not a test case can be safely excluded.

While TestTube resembles DejaVu and Pythia in many ways, there are also many structural differences. One significant difference is that TestTube does not explicitly consider control-flow order: test traces list functions called and global entities potentially referenced, but with no particular order imposed. While this can be viewed as just an approximation to what DejaVu and Pythia do, it is appropriate to view TestTube in its own right. A SafeSpace instantiation for TestTube, then, should reflect TestTube's unique qualities.

First, in a SafeSpace instantiation $I_T$ of TestTube, the set of semantic words can contain the checksum ids for functions, macros, type definitions, and global variables — everything in a program's global namespace. TestTube's models of the state and agent set parallel those of DejaVu and Pythia. The traces for this SafeSpace proof resemble regular TestTube traces: they consist of a set of unordered semantic words. Unlike TestTube, however, an extra symbol "End" is added to the end of each trace, where all other entities in the trace are partially ordered such that they must occur before "End." This last symbol is the only ordered member of the trace.

Given that TestTube does not explicitly consider control-flow order, we will use mostly unordered graphs for the proof. Specifically, $\delta$ maps semantic words in the programs to nodes in no particular order. The $\epsilon$ function then joins these nodes into a complete directed graph, as shown in Figure 8. Edges in this graph are labeled with the value of their destination. The "Start" node is then added and connected to all other nodes in the graph. Edges leading from the "Start" node to other nodes are labeled with the id of the destination

\(^{15}\)Note that "change" here actually denotes cases where the entity is added, deleted, or modified.
Figure 8: Left: three function declarations and a macro (labeled with letters); right: graph representation of program

node. Finally, an “End” node, which corresponds to the extra semantic word placed in the execution traces, is added, and directed edges are added from this node to all nodes other than the “Start” node.

During the execution of SafeSpace on $G$ and $G'$, any changed node $N$ will cause all edges $(X,N)$ to be diverted to the Select node; following execution, all traces containing changed entities will necessarily end up in the Select node of $G_{sig}$, no matter how we order the entities of the traces (except of course, that “End” must be last).

Although we do not provide a proof here, the regression bias of this SafeSpace instantiation $I_T$ necessarily contains the bias of DejaVu. This is the case because both DejaVu and TestTube have been shown to select all modification-traversing test cases [21], and TestTube has been further shown to select all test cases selected by DejaVu. Because TestTube does not explicitly consider control flow, TestTube's bias might be non-trivially larger than the biases of DejaVu and Pythia: as we noted in Section 3, however, this only means the set of programs over which TestTube is safe is potentially larger, but it does not in and of itself make TestTube a more effective technique.\(^{16}\)

5.3 Utilizing the Safe RTS Framework to Create a New Algorithm

So far, we have used the safe RTS framework to prove RTS algorithms safe. We now consider another use of the framework: facilitating the creation of new safe RTS algorithms. As an example, we propose a new RTS technique designed for spreadsheet languages, rather than the imperative programs addressed by DejaVu, Pythia, and TestTube.\(^{17}\) To the best of our knowledge, this new RTS technique is the first safe RTS technique targeting programs outside of the deterministic, imperative regression bias. By demonstrating the use of our framework in this context, we provide evidence that the framework is generally applicable.

\(^{16}\)In fact, previous studies by the authors have shown that TestTube can cost either more or less than DejaVu, depending on details of the subject programs and test suites [7, 21].

\(^{17}\)This algorithm is particularly interesting in that the framework it creates is widely applicable to declarative languages, although the particular template functions used would be inappropriate given that different sets of requirements govern graphics content languages. For example, like spreadsheet languages, VRML has nodes that contain geometry and links that route messages between nodes. This structure suggests that the partial-order approach we use for spreadsheet languages would be useful for VRML, even though VRML has many complications have not been considered for the spreadsheet algorithm (for instance, node hierarchies, infinite message loops, and script nodes potentially not meshing with the declarative paradigm [4, 9].
Creating a safe RTS algorithm for spreadsheet languages is also, however, a useful goal in and of itself.

Techniques for testing programs in spreadsheet languages differ from techniques for testing programs in imperative languages. Some of these differences will be discussed in this paper; however, we refer the reader to other references for more detail [23, 27, 28].

5.3.1 The spreadsheet language paradigm (using Forms/3)

The term “spreadsheet languages” refers to a class of declarative visual languages that interlink sets of formulas to generate outputs; usually, only the final outputs are written to the screen — intermediate calculations are performed invisibly. Since spreadsheet languages can potentially express a wide range of behaviors, we have chosen to consider, as a representative of these languages, a subset of the visual research programming language Forms/3 [28]. Table 1 presents the grammar for this subset. The subset that we have chosen represents the behavior found in typical spreadsheet applications.

Figure 9 depicts a Forms/3 program containing several cells, with their formulas displayed; this program calculates the root(s) of quadratic and linear equations. As the figure illustrates, unlike more “traditional” spreadsheet languages, Forms/3 does not restrict cells to a grid pattern. To create such a program, users place formulas in the individual cells, and the system responds to edits by updating other cells (or at least, other visible cells) that depend on those cells. Formulas in cells can access the results of other cells, thus imposing a causal ordering on the set of nodes that must be satisfied for correct computation. In general there are many possible sequences of formula evaluations leading to correct results, and the evaluation engine running beneath the scenes is free to choose any such order, provided that the data dependencies between cells are maintained. All correct Forms/3 programs are guaranteed to halt due to the absence of cycles in the causal ordering.

The testing process model for the spreadsheet language used in this paper is based on a model presented in [28]. The execution of a test case on a spreadsheet involves placing test values in input cells (cells whose formulas are constant) and then checking query cells (output cells of interest) after the program has updated itself, to determine whether those cells contain proper values. A particular query cell contains a mini-program that computes a formula; formulas potentially depend on the values of several other cells. Thus, before a cell (query or otherwise) can be evaluated, the cells upon which that cell is data dependent must be evaluated. The evaluation engine keeps track of which cells have been changed and therefore which cells need to be updated. In reverse, we can view this process as the query cell prompting its ancestors to
evaluate themselves, where the ancestors might then need to prompt ancestors of their own.

Rothermel et al. have specified a graph representation for modelling spreadsheet programs, called the cell relation graph (CRG) [28]. Figure 10 depicts the CRG for the program of Figure 9. The cell relation graph represents each individual cell (and thus, the formula for that cell) in a Forms/3 program as a node. Directed data dependence edges connect each node to all nodes whose results the node immediately requires for its own internal evaluation. The result is a directed acyclic graph (DAG).

In terms of the cell relation graph, the foregoing testing process model can be viewed as creating a subset DAG of the original DAG of the cell-relation graph. Following the application of a set of test inputs to a Forms/3 program, and the identification of a query cell, the subset DAG created by that test necessarily includes at least one node corresponding to a cell that can be evaluated immediately. Additional cells (nodes) then continue to be evaluated until the specified query node can be evaluated. We term the set of nodes involved in the computation of the value for a given query node an execution DAG.

5.3.2 Using the framework to create a new safe RTS algorithm

The sequence of steps we follow to generate a safe RTS technique $A$ parallel the steps involved in proving an RTS technique safe:

**Five-step algorithm generation process:**

1. Based on the requirements of the testing process, propose a regression bias $A_B$ for $A$.
2. Propose a model of the state $S$ and semantic word set $\Sigma$, both satisfying the stable semantics condition (SSC) when $A_B$ holds. At the same time, describe the $\mu$ and $\neg\mu$ agent sets, as well as indicators that detect when $\neg\mu$ events occur if $\neg\mu$ is nonempty.
3. Construct $\delta$ and $\epsilon$ functions out of the state model $S$ and semantic word set $\Sigma$ such that the graphs so produced capture $A_B$.
4. Transform trace information into abstract execution traces for SafeSpace.
5. Construct an $\iota$ function that guarantees safety whenever $A_B$ holds.

Note that this process can be iterative. While we may know in advance the specification of a regression bias, the best model of semantics or graph structure might not be obvious. Further, repeated iterations might suggest that the initial regression bias can, in fact, be refined and still meet our testing requirements, thus potentially allowing for substantially more efficient algorithms without loss to safety.

**Step 1: Propose a regression bias**

The creation of a regression bias for a new technique is little different than the creation of a SafeSpace instantiation for an existing technique. In both cases, there exists a set of programs on which the safe RTS technique is to operate. The regression bias merely specifies the set of attributes possessed by this set of programs, both positive and negative, as well as any constraints on test cases necessary to preserve the bias.

We begin our construction of a regression bias for spreadsheet languages quite simply: we are interested in spreadsheet programs that follow Alan Kay’s value rule, which states that the value of a cell is defined solely by the formula explicitly specified for that cell [17]. To be more precise, we are targeting the subset
Figure 9: Forms/3 program to obtain the root(s) of quadratic and linear equations.

Figure 10: Cell relation graph for the Forms/3 program of Figure 9.
of Forms/3 programs described in Section 5.3, which clearly satisfies this value rule. The program in Figure 9 is an example of such a program.

To further refine our bias, we need to consider the expectations and needs of the users of our safe RTS technique. Specifically, we are interested in augmenting a spreadsheet programming environment for everyday end-users — thus our testers will likely not be professional testers, nor will they have any interest in understanding testing theory. Such users expect that if they hit a “help me retest this” button, the result will be fairly fast and will be safe. (Of course, such testers are unlikely to understand the distinction between safe and unsafe RTS techniques, but a user’s expectation that the system will perform a best-effort attempt at error detection essentially implies safety.) We could employ the retest-all approach, but that potentially violates the rapid-response goal. We thus want to find an efficient safe RTS technique.

Given our definition of the expected tester, we need to further clarify our bias given our assumptions about the relation of the programmer and the programming environment. First, since our testers are end-users of a software product, they are not expected to fix faults in the spreadsheet environment itself. Our technique, therefore, assumes that the spreadsheet environment’s evaluation engine performs correctly. Essentially, this is equivalent to DejaVu, Pythia, and TestTube’s assumption that the compiler operates correctly. Note, however, that DejaVu and the other algorithms must make further equivalence assumptions about other factors in the external environment — for example, that the operating system is identical between runs. The spreadsheet evaluation engine provides a front end for all of these factors that is transparent to the bias, since correct execution of the engine requires that these other aspects hold. Correct execution of the evaluation engine also incorporates another assumption: all programs halt. However, given the simple nature of spreadsheet languages, verifying that a program halts requires only that we check for cycles in cell references. Finally, our bias assumes that $P$ is correct for $T$: since the program has been executed previously, we presume that faults previously made visible by $T$ have been detected and corrected.

Altogether, these assumptions form a regression bias $R$ for our new algorithm:

**Regression Bias for Spreadsheet Languages** — The underlying environment executes correct Forms/3 programs (that possess the constructs outlined above) according to the specifications of the language, and the original program $P$ is correct for $T$.

**Step 2: Propose state model, semantic word set, and agent sets**

Given our regression bias, our state $S$ consists solely of the values assigned to the cells of the spreadsheet, including formulas, input values, and cells changed in the process of evaluation. The transient state $X$ consists of all cells not contained in the execution DAG for the query cell. In this case, $X$ is a well-defined set of values, primarily because our language is purely declarative with no side effects.

Our semantic word set consists of the cell formulas in the spreadsheet program. This can, of course, involve words that correspond to requests for user input. The valid agent set consists of the set of correct behaviors associated with the evaluation engine. Because the evaluation engine shields the user program from the outside environment, all rogue agents result only from incorrect behavior in the evaluation engine (although incorrect behavior of the engine could arise from external events).
Step 3: Construct $\delta$ and $\epsilon$ functions

The $\delta$ and $\epsilon$ functions for our algorithm create CRGs. The $\delta$ function is quite simple for this algorithm: semantic words are related one-to-one to formulas in individual cells. The $\epsilon$ function creates directed edges between cells that contain sources and uses of variables, similar to those found in the CRG; however, for our algorithm, we reverse the direction of these edges: this change makes the graph-walk procedure simpler. Although individual cells can contain small programs themselves, we have chosen not to consider the internals of cell formulas in this algorithm, in the interest of maintaining a rapid response time. Our $\epsilon$ function thus forms a DAG out of the nodes created by $\delta$, just as the cell-relation graph is itself a DAG.

Step 4: Transform trace information into abstract execution traces

As described earlier, the execution of a test case for a Forms/3 program involves applying input values and then selecting a query cell. Obviously, the application of input values can only change parts of the graph that are dependent on the changed values. The evaluation engine need only check those parts of the graph.

Thus, an execution trace for a test case is a subgraph $G''$ of $P$'s graph formulation $G$ (which is itself a coarse version of $P$'s CRG). We can represent such a trace simply as a bit vector with 1's for all nodes contained in the execution trace. Note that these traces represent partial orders over cells in a graph.

Step 5: Construct the $\iota$ function

In proofs of safety, the $\delta$, $\epsilon$, and $\iota$ functions are intrinsically tied together; a $\delta$ function may be chosen such that a convenient $\iota$ function exists. In the algorithm creation process, however, we have chosen to separate the $\delta$ and $\epsilon$ function creation step from the $\iota$ function creation step, because here, we wish to emphasize modeling of the program before concerning ourselves with proving equivalence. Of course, difficulties in the creation of a valid $\iota$ function can lead to a reevaluation of the $\delta$ and $\epsilon$ functions created previously.

In this case, given the $\delta$ and $\epsilon$ functions, there is a clear choice for the $\iota$ function. Just as with DejaVu, we could use lexical equivalence as our $\iota$; such a choice is simple and possesses clear semantics. However, we are primarily interested in creating a fast safe RTS algorithm, in interest of maintaining the rapid response time demanded by users of spreadsheets. Thus, instead of having $\iota$ determine lexical equivalence between cell formulas, we will keep an extra bit of information for each cell to indicate whether that cell's formula has been modified. These bits are set to “1” when the cells’ formulas are first entered, and set to “0” after the user has validated the cell by exercising its incoming dependencies [28] executed the safe RTS function. If a cell's formula is subsequently altered, its bit is reset to “1". Thus, our $\iota$ function needs merely to shift bits out of two bit strings to detect “inequivalence”. If the bit vector representing changes contains a 1 where the bit vector representing the execution DAG for a test case contains a 1, that test case needs to be rerun.

Up to this point, we have not shown that our chosen $\delta$, $\epsilon$, and $\iota$ functions suffice as structure and equivalence functions in SafeSpace: because the proof is rather simple we have delayed it until $\iota$ was defined. $\delta$ and $\epsilon$ clearly capture the semantic behavior of spreadsheet programs: their relation to CRGs guarantees this. The key to the proof thus lies entirely with the $\iota$ function. As we stated earlier, an $\iota$ function must guarantee that given identical start states, equivalent nodes under $\iota$ produce equivalent non-transient states; the $\iota$ function given here clearly achieves this goal as it indicates equivalence only when the node has not been altered by the user, and thus represents the same local function. From a more global perspective, we
see that this algorithm is safe because it excludes only test cases whose trace in \( P \) contains no nodes that have been changed to create \( P' \). Because new edges can only arise when nodes are changed, if no nodes in a trace have changed, we know that the test case has an equivalent execution DAG in both \( P \) and \( P' \). Thus, the behaviors of \( P \) and \( P' \) on that test case are necessarily equivalent.

5.3.3 Test Selection: a Safe Heuristic

The SafeSpace algorithm focuses on constructing a data structure that is used in selecting dangerous edges. We have not yet discussed how test case selection is performed, given those edges, by our spreadsheet algorithm. Spreadsheet languages have some interesting traits that make this selection efficient.

As we stated earlier, there are two levels of granularity in most spreadsheet languages: inter-cell dependencies and formulas inside of cells. Our algorithm focuses on inter-cell dependencies. As such, a particular trace as defined for our technique might apply to a large number of input-output relations. For instance, there are only two distinct traces for the quadratic equation solver, and yet there are many possible test cases required to test boundary conditions and ensure the correct computation of roots by that program. In more complex spreadsheet programs with less interconnected graphs, there can be a large number of distinct traces; however, it is still often the case that there are far more test cases than unique test traces. Thus, we can “bin” test cases before the algorithm is run to create a more efficient implementation. We keep in storage a list of all unique test traces seen previously; as we add test cases, we check to see if the trace of each test case is unique. If the trace is unique, we add it to the list; if the trace is not unique, we add that test case to a list associated with the trace. After we run the SafeSpace implementation for spreadsheet languages we run our test selection process on the traces rather than the test cases: in this way, we avoid checking a trace for each test case, and potentially reduce our work significantly. Clearly, this choice is safe. For programs with substantial numbers of test cases executing identical formulas, this optimization can produce substantially greater efficiency.

5.3.4 Example of the Application of the Algorithm to a Forms/3 Program

Figure 11 presents a simple Forms/3 program. Because we cannot clearly represent input cells as specified in the visual format of Forms/3, we have put “read” primitives in cells that require user input. This program does not compute any meaningful functions, but it suffices to demonstrate our RTS algorithm.

Alongside the Forms/3 program is its graph representation as specified by the \( \delta \) and \( \epsilon \) functions previously described. As we noted in Section 5.3.2, this representation depicts inter-cell dependencies, except that the directions of the arrows have been reversed. Thus, edges from a node point to all nodes whose associated cells are referenced in that node’s local formula. The nodes in the graph have been labeled with the names of the cells they represent, as defined by the Forms/3 program.

As an example, we consider two test cases. The first is a query on only variable \( f \), the second is a query on only variable \( g \). As we discussed, traces include all edges traversed in the course of a computation of the value of the query variable. Thus, the trace for test case 1 (on \( f \)) contains edges \( a_1, a_2, b_1, b_2, d \), and \( f \). The trace for test case 2 (on \( g \)) contains edges \( a_4, b_3, c_1, c_2, e, \) and \( g \).

Suppose that a modified version of this program has been created. This version \( P' \) is presented in Figure 12 with its graph representation \( G' \). To generate this program, we added a new cell \( x \) and referenced that cell in the formula for \( f \). Thus, all nodes other than \( x \) and \( f \) have their change bit set to 0; \( x \) and \( f \) have their
a  read a
b  read b  d  if (a > b) then 0 else 1
c  read c  e  if (c < b) then (a*a) else (b*b)
f  if (d = 0) then "Invalid Entry" else (b / a)
g  if (e > c) then "Invalid Solution" else (e / c)

Figure 11: Left: a Forms/3 program; right: the graph representation $G$ of the program.

a  read a  x  if ((a*b - c) = 0) then 1 else 0
b  read b  d  if (a > b) then 0 else 1
c  read c  c  if (c < b) then (a*a) else (b*b)
f  if (d = 0) then "Invalid Entry" else if (x = 1) then a else (b / a)
g  if (e > c) then "Invalid Solution" else (e / c)

Figure 12: Left: version 2 of the Forms/3 program; right: the graph representation $G'$ of this version.

Select

a

b

c

d

e

Figure 13: The signature graph $G_{sig}$ of $G$ and $G'$. 
change bit set to 1. We then execute the SafeSpace instantiation as described previously on graphs \( G \) and \( G' \), producing \( G_{\text{sig}} \) as shown in Figure 13. Note that, because \( f \) was altered to create \( P' \), the arrow from Start to \( f \) is directed immediately to Select. Thus, we can immediately see that test case 1, which contained edge \( f \), should be selected for retest, while test case 2 need not be re-validated.

One of the main advantages of the safe RTS technique presented here is that it can easily be incorporated into a testing environment such as the environment described in [28]. A reasonable implementation of this algorithm in that environment would execute the safe RTS algorithm, run the required test cases, and highlight the query variables requiring revalidation. This would direct the user’s attention to the values that might have been changed as a result of changes. Thus, safe RTS not only increases the responsiveness of a revalidation command, but also focuses a user’s attention on potential trouble spots.

6 Conclusions

In the past two decades, over twenty distinct regression test selection algorithms have been proposed, utilizing a wide range of approaches (see [25] for a list). Yet in all this work, only two papers [5, 25] can be said to consider regression test selection from a more theoretical standpoint. This paper attempts to extend our understanding of the theoretical foundations of regression test selection, focusing on safe RTS techniques.

In establishing a framework for safe RTS, we establish what is, essentially, a specification for how a safe RTS algorithm should be designed, and clarify the contracts that safe RTS testing must make, thereby allowing software testers and clients to more readily agree as to what testing standards have been met. We also believe that this framework can make the task of designing a new safe regression test selection algorithm easier for a researcher or test engineer in need of a safe RTS solution for a unique testing scenario. In this context, we showed that safe RTS algorithms can be designed almost entirely by only slotting functions in the generalized safe RTS algorithm. Our emphasis on the process of the technique was intended to further facilitate the practical usefulness of the framework.

So far, however, we have not discussed the place of safe RTS testing in the global scheme of software engineering. In Section 2, we suggested that the purpose of safe RTS testing was just to save money, or somewhat equivalently, time; this is indeed a valid use of safe RTS techniques and, in the past, the primary justification for their use. Nonetheless, safe RTS techniques represent only one branch of regression testing philosophies. Safe RTS algorithms operate on the assumption that test cases are associated with faults and thus, that the goal of testing is only to find faults. There is nothing wrong with this goal, and if safe RTS algorithms can help us find faults at less expense, then clearly it would be foolish not to use them. On the other hand, finding faults is not the only means of improving software quality, and possibly, finding faults it not even the best approach.

Safe RTS algorithms have deeper uses, however, in the maintenance phase of the software life cycle. RTS algorithms, in general, let us partition our testing resources more efficiently with a little advance work. By defining a “safe space” in the code, safe RTS techniques let us exclude portions of code from further testing with fault-finding techniques because their usage tells us that those domains are behaviorally equivalent. Thus, coupled with adequate test suites, safe RTS techniques can provide justification that faults are not likely to be located in excluded code. This can give testers a measure of the utility of performing different types of additional testing. Having employed a safe RTS technique, a tester knows that executing tests over
the excluded section of the program will likely not be of much use. A tester can then allocate more resources to other activities such as reliability assessment, or increasing test coverage on areas known to have high complexity measures.

Consider that DejaVu, for example, has been shown to reduce the cost of retesting by 75-88% on the 50,000 line program Empire[22]. Further, the technique displays a tendency to produce greater savings as programs grow larger. The chunk of regression testing time saved can be reallocated to completely different validation processes. On the other hand, while unsafe RTS techniques perform a function similar to that of safe RTS techniques, they do not, in general, justify the same confidence levels. Thus, while an unsafe technique might cut the size of the test suite by a significant amount, we might not be justified, as we are with safe techniques, in instituting processes of testing over other domains, because we have not adequately tested the code of greatest concern. Clearly, safe RTS algorithms belong in a tester’s toolbox as they provide analytic guidance on how to increase the quality of software with limited resources.

Of course, primarily, safe RTS techniques are intended to provide safe savings in software engineering costs. As evidenced in experiments performed by Rothermel and Harrolld [24], Chen, Vo, and Rosenblum [11], and Vokolos and Frankl [30], such savings can be made for realistic programs without missing faults. As testing methods mature and as the demand for post development testing increases, it seems a given that safe RTS analysis methods will be fundamental for later-life software maintenance.

7 ACKNOWLEDGEMENTS

This work was supported in part by a grant from Boeing Commercial Airplane Group, and by NSF Faculty Early Career Development Award CCR-9703185 to Oregon State University. We also wish to acknowledge S. S. Ravi and Dan Rosencrantz for enlightening discussions on safe RTS.

References


