Automatically and Transparently Distributing the Execution of Regression Test Suites

Gregory M. Kapfhammer
Department of Computer Science
Allegheny College
gkapfham@allegheny.edu

May 31, 2001

Abstract

From the perspective of the software testing practicioner, regression testing is often an important technique for developing high quality software systems. The software testing research community has developed several different approaches in order to ensure that regression testing process is more cost-effective and practical. The techniques developed by Harrold, Rothermel, Rosenblum, Weyuker, Soffa and others have attempted to increase the cost-effectiveness of regression testing by intelligently selecting a subset of an entire regression test suite for execution. Recent approaches created by Elbaum, Malishevsky, Rothermel, Untch, Chu, and Harrold have tried to improve a regression test suite's rate of fault detection by prioritizing the execution of the tests. In this paper, we describe the conceptual foundation, design, and implementation of an approach that distributes the execution of regression test suites. We show that our technique can complement existing regression test selection and prioritization approaches. We also demonstrate that it can be used independently when existing approaches are not likely to increase regression testing efficiency. Finally, we describe the design and implementation of `Joshua`, an application that facilitates the distributed execution of regression test suites for Java-based software systems. [1]

1 Introduction

After a software system experiences changes in the form of bug fixes or additional functionality, software regression testing is used to determine if these changes introduced new defects. The creation, maintenance, and execution of a regression test suite helps to ensure that the evolution of an application does not result in lower quality software. The industry experiences noted by Onoma et al. indicate that regression testing often has a strong positive influence on software quality [19]. Indeed, the importance of regression testing is well understood. However, Beizer, Leung, White and countless others have noted that many software development teams might choose to omit some or

[1] Information about `Joshua` is available at `http://cs.allegheny.edu/~gkapfham/research/joshua/`.

all regression testing tasks because they often account for as much as one-half the cost of software maintenance [4, 17]. Moreover, the high costs of regression testing are often directly associated with the execution of the test suite. Other industry reports from Rothermel et al. show that the complete regression testing of a 20,000 line software system required seven weeks of continuous execution [26]. Since some of the most well-studied software failures, such as the Ariane-5 rocket and the 1990 AT&T outage, can be blamed on the failure to test changes in a software system, it is clearly important to develop and refine techniques for efficient regression testing [14].

Many different methods have been developed in an attempt to reduce the cost of regression testing. Regression test selection approaches attempt to reduce the cost of regression testing by selecting some appropriate subset of the existing test suite [3, 23, 29]. Test selection techniques normally use the source code of a program to determine which tests should be executed during the regression testing stage [22]. Test case prioritization techniques attempt to order a regression test suite so than those tests with the highest priority, according to some established criterion, are executed earlier in the regression testing process that those with lower priority [11, 25]. By prioritizing the execution of a regression test suite, these methods hope to reveal important defects in a software system earlier in the testing process.

In this paper, we explore the regression testing problem and successively build a solution to this problem that relies upon test selection, prioritization, and distribution. In Section 2 we revisit the definition of the regression testing problem and explain current solutions to the challenges normally associated with it. Section 2.2 contains a description of Rothermel and Harrold's algorithm that uses test selection to increase regression testing practicality. Section 2.3 presents an algorithm proposed by Rothermel, Untch, Chu, and Harrold that can rely upon both test selection and prioritization to solve the regression testing problem. Section 3 offers a formulation of our approach to distributing the execution of a regression test suite. We discuss scenarios for which the assumptions made by current regression testing methodologies might be insufficient or unrealistic. We also demonstrate how our approach can be used in conjunction with or in isolation from other regression testing techniques.

In Section 4 we examine the considerations that must be addressed in order to develop an application that reliably and efficiently distributes the execution of a regression test suite. Section 5 explains some of the distributed computing concepts that are central to our technique for the improvement of regression testing cost-effectiveness. We also explore the design and implementation details of `Joshua`, an application that performs distributed regression testing. Finally, Section 6 concludes with future research and development avenues to improve our approach to automatically and transparently distributing the execution of regression test suites.

2 Regression Testing Background

In this section we will review the notation that is commonly used to define and discuss the regression testing problem. Furthermore, we will characterize two existing solutions to the problem of practical and cost-effective regression testing. The review of the existing regression testing tech-

niques that have been developed by Rothermel, Harrold, and others will contextualize and motivate the discussion of our approach to regression testing.

2.1 Notation

For simplicity, we adopt the regression testing notation popularized by Rothermel and Harrold and used throughout [22]. We will briefly summarize this notation and extend it as needed throughout the remainder of this paper. We let P be a program and P' be a modified version of P. We use S to denote the specification for program P and S' to denote the specification for program P'. Furthermore, we let $P(i)$ refer to the output of P on input i and $P'(i)$ refer to the output of P' on input i. In accordance with the established notation, we define a test to be a 3-tuple $<$ *identifier, input, output* $>$, where *identifier* is the name of the test, *input* refers to the input values for the given test, and *output* is the expected output for the test given by $S(input)$. Finally, we will use $T = \{t_1, \ldots, t_n\}$ to denote a test suite or test set. The regression testing problem was characterized by Rothermel and Harrold in the following fashion [21]:

Problem 1 (Regression Testing) *Given a program P, its modified version P', and a test set T that was used to previously test P, find a way to utilize T to gain sufficient confidence in the correctness of P'.*

2.2 Selective Retest Techniques

Selective retest techniques attempt to reduce the cost of testing by identifying the portions of P' that must be exercised by the regression test suite. Selective retesting is distinctly different from a retest-all approach that always executes every test in an existing regression test suite. We will use the notation *RTS* to denote a solution to the regression testing problem that relies exclusively upon test selection. Rothermel and Harrold have also characterize the typical steps taken by *RTS* in the following manner [22, 24]:

Algorithm 1 (Regression Testing with Selection)

1. *Select $T' \subseteq T$, a set of tests to execute on P'.*
2. *Test P' with T' in order to establish the correctness of P' with respect to T'.*
3. *If necessary, create T'', a set of new functional or structural tests for P'.*
4. *Test P' with T'' in order to establish the correctness of P' with respect to T''.*
5. *Create T''', a new test suite for P', from T, T', and T''.*

Each one of the steps in Algorithm 1 addresses a separate facet of the regression testing problem. Step 1 attempts to select a subset of T that can still be used to effectively test P'. Step 3 tries

to identify portions of P' that have not been sufficiently tested and then seeks to create these new regression tests. Step 2 and Step 4 focus on the efficient execution of the regression test suite and the examination of the output for incorrect results. Finally, Step 5 highlights the maintenance activities that are often associated with regression testing. Regression test selection mechanisms limit themselves to the problem described in Step 1. For more details about safe regression testing and the assumptions under which it is most beneficial, please refer to [21, 22]. In Section 3 we highlight the strengths and weaknesses of the regression test selection method and show how it can be used in conjunction with an approach that distributes the execution of a regression test suite.

2.3 Test Prioritization Approaches

Regression test prioritization approaches assist with regression testing in a fashion that is distinctly different from test selection methods. Test case prioritization techniques allow testers to order the execution of a regression test suite in an attempt to increase the probability that the suite might detect a fault at earlier testing stages [11, 25, 32]. It has been noted that test case prioritization approaches can be used in conjunction with regression test selection methods since they do not discard test cases [26]. We will use the notation *RTSP* to denote a solution to the regression testing problem that might rely upon test selection and/or test prioritization. Using the formalization of Elbaum, Malishevsky, Rothermel, and Harrold we can characterize the typical steps taken by *RTSP* in the following fashion [11, 22]:

Algorithm 2 (Regression Testing with Selection and Prioritization)

1. *Select $T' \subseteq T$, a set of tests to execute on P'.*
2. *Produce T'_p, a permutation of T', such that T'_p will have a better rate of fault detection than T'.*
3. *Test P' with T'_p in order to establish the correctness of P' with respect to T'_p.*
4. *If necessary, create T'', a set of new functional or structural tests for P'.*
5. *Test P' with T'' in order to establish the correctness of P' with respect to T''.*
6. *Create T''', a new test suite for P', from T, T'_p, and T''.*

Note that *RTSP* provides a solution to the regression testing problem that relies upon both test selection and test prioritization. The expression of Algorithm 2 is intended to indicate that a tester can use regression test selection and prioritization techniques in either a collaborative or independent fashion. Step 1 allows the tester to select T' such that it is a proper subset of T or such that it actually contains every test that T contains. Furthermore, Step 2 could produce T'_p such that it contains an execution ordering for the regression tests that is the same or different

than the ordering provided by T'. The other steps in Algorithm 2 are similar to those outlined in Algorithm 1, the regression testing solution that relied only upon test selection.

The test case prioritization approaches developed by Elbaum et al. restrict themselves to the problem described in Step 2. Testers might desire to prioritize the execution of a regression test suite such that code coverage initially increases at a faster rate. Or, testers might choose to increase the rate at which high-risk faults are first detected by a test suite. Alternatively, the execution of a test suite might be prioritized in an attempt to ensure that the defects normally introduced by competent programmers are discovered earlier in the testing process. Current techniques that prioritize a regression test suite by the fault-exposing-potential of a test case also rely upon the usage of mutation analysis [11].

The mutation analysis technique determines whether a test suite is able to detect the defects in a finite set of programs that are approximately correct [9, 14, 15]. When a regression test suite for a given program P is subjected to a mutation analysis, a finite set of syntactically different versions of P are produced. Then, the entire regression test suite is executed for each one of the mutated versions of P in order to determine if the tests can detect the fault that each mutant represents. Even though mutation analysis has proven to be a viable regression test prioritization technique, its practicality is limited since it is so computationally intensive [11]. In Section 3 we highlight the strengths and weaknesses of the regression test prioritization method and explain how it can be used in conjunction with our approach to regression test suite distribution.

3 Distributed Regression Test Execution

In this section, we will explain an alternative technique for improving the cost-effectiveness of regression testing. We will also examine the challenges that are faced by the other solutions to the regression testing problem. Finally, we will identify ways in which the distributed execution of regression test suites can meet the challenges that are incompletely handled by other approaches.

3.1 An Alternative Approach

The process of distributing the execution of a regression test suite attempts to reduce the cost of testing by more fully utilizing the computing means that are normally available to a testing team. Our technique exploits the computational resources that existing development machines provide in order to ensure that regression testing is more cost-effective and practical. The distributed execution of a regression test suite might ensure that it is practical to test a software system that could previously not be tested in a cost-effective fashion. Moreover, distributing the execution of a regression test suite should only serve to increase the efficiency by which we can test software systems with other regression testing approaches. We use the notation *RTSPD* to denote a solution to the regression testing problem that can rely upon selection, prioritization, and distribution mechanisms. Building upon the formalization developed by Elbaum et al. and Rothermel et al. we can characterize the steps taken by *RTSPD* in the following manner [11, 22]:

Algorithm 3 (Regression Testing with Selection, Prioritization, and Distribution)

1. *Select $T' \subseteq T$, a set of tests to execute on P'.*
2. *Using m machines, produce T'_p (a permutation of T'), such that T'_p will have a better rate of fault detection than T'.*
3. *Using m machines, test P' with T'_p, in order to establish the correctness of P' with respect to T'_p.*
4. *If necessary, create T'', a set of new functional or structural tests for P'.*
5. *Using m machines, test P' with T'', in order to establish the correctness of P' with respect to T''.*
6. *Create T''', a new test suite for P', from T, T'_p, and T''.*

Note that *RTSPD* offers a solution to the regression testing problem that relies upon test selection, prioritization, and distribution. Our expression of Algorithm 3 is intended to indicate that a tester can use selection, prioritization, and distribution in either a collaborative or independent manner. Step 2 assumes that m machines can be used during the analysis needed to create an appropriate permutation of the regression test suite T'. Step 3 and Step 5 require the usage of some mechanism to distribute the execution of an entire regression test suite across m machines. Therefore, the distribution technique employed by *RTSPD* can be used in three separate steps in an attempt to make regression testing more practical. The remainder of the steps taken by the solution that *RTSPD* provides are the same as those outlined in Algorithm 2. In order to better understand the importance of distributed regression test suite execution, we must examine the strengths and weaknesses of the selection and prioritization techniques included as part of Algorithm 3.

3.2 Challenges and Solutions

The solution provided by *RTSPD* can rely upon a regression test selection approach. However, Rothermel and Harrold indicate that the assumptions that form the basis for all safe regression test selection mechanisms make it difficult to perform any test selection if the changes that affect P involve only P's operational environment [22]. However, the position of some software experts indicates that changes in the operational profile of a software system are likely to affect the execution of the system [30]. For example, suppose that system P was developed with the Java programming language and it is known to run correctly with version 1.1 of the Java virtual machine. If P' is produced by changing the operational environment of P so that it now executes on version 1.3 of the Java virtual machine, it will be challenging for test selection techniques to improve the efficiency of regression testing.

The regression test selection technique provided by *RTSPD* makes no assumptions about the types of tests that exist in the regression test suite T. In fact, test selection approaches assume

that every test within T is subject to elimination. However, the industry experience described by Onomal et al. indicates that the test cases that are involved with application environment simulation or user acceptance testing are normally not considered candidates for exclusion [19]. For example, suppose that software developers create a collection of regression tests that simulate a real world application environment and then use these tests to measure the quality of subsequent releases. Or, consider a regression test suite that is composed of tests that detect defects that were discovered by customers through actual field usage. If either of these situations arise in a typical industry software development environment, test selection techniques will not be able to improve regression testing cost-effectiveness because no proper subset of these regression test suites can be selected [19].

RTSPD provides a solution to the regression testing problem that can also rely upon test case prioritization. The current empirical analyses conducted by Elbaum et al. and Rothermel et al. indicate that prioritizing the execution of a regression test suite can be very useful [11, 25]. However, it is often computationally expensive to order a regression test suite's execution based on code coverage levels or fault-exposing-potential. For example, suppose that T contains a large number of tests and mutation analysis is used to produce many mutant versions of P. In order to calculate the fault-exposing-potential for each test $t \in T$, t must be executed against each of P's mutants and the output must be compared to P's output. In this situation, the practicality of the regression testing process might be negatively impacted if test case prioritization is used.

The distribution mechanism provided by solution *RTSPD* can be used to assist in the previously outlined scenarios. If the changes that are applied to P to produce P' involve the program's environment, our distribution mechanism can be used to increase regression testing efficiency. Furthermore, if it is not possible to apply test selection to a large subset of tests that simulate an application environment or represent defects found in the field, distributed test execution can remedy this concern. When the computational requirements of test case prioritization are particularly daunting, our distributed test execution approach can be used to make prioritizations based on coverage-levels or fault-exposing-potential more practical. In situations where test selection and/or prioritization are possible, the distributed execution of a regression test suite can be used to further enhance the cost-effectiveness of the testing process. When only a single test machine is available for regression testing, the distribution mechanism can be disabled and the other testing approaches can be used to solve the regression testing problem.

4 Test Distribution Considerations

The solution that *RTSPD* provides to the regression testing problem can rely upon the distributed execution of a test suite. Any tool that attempts to distribute the execution of a test suite must evaluate a number of implementation considerations in order to ensure that this efficiency enhancement does not negatively impact the reliability of the regression testing process. Suppose that $T = \{t_1, \ldots, t_n\}$ and there are m machines that are available for regression testing. The following

considerations must be taken into account in order to reliably improve the cost-effectiveness of regression testing:

Consideration 1 (Transparent and Automatic Distribution) *When desired, the distribution of the n tests cases across the m machines should be as transparent and automatic as possible.*

Consideration 2 (Test Case Contamination Avoidance) *When the n test cases are distributed across the m machines, it must be done in a manner that attempts to avoid test case contamination.*

Consideration 3 (Test Load Distribution) *When the n test cases are distributed across the m machines, it must be done in a manner that appropriately distributes the load during the test execution process.*

Consideration 4 (Test Suite Integrity) *The process of distributing the n test cases across the m machines should not affect the correctness of test case results or impede the proper execution of the test suite.*

Consideration 5 (Test Execution Control) *It must be possible to control the execution of the m test cases and view the results of each test from a centralized testing environment.*

Consideration 1 states that the parties who are involved with regression testing should be minimally responsible for test case distribution. Thus, the tester should not be required to explicitly configure most facets of the distribution process. For example, a tester should not be forced to describe the directory structure of a given program's source or ensure that program executables are available on each of the test execution machines. However, it might be appropriate for the tester to initially specify the names of m machines that are available for regression test suite execution.

The important issue of test case contamination is addressed in Consideration 2. A test case is considered to be independent if its success or failure does not depend upon the order in which the entire regression test suite is executed [18, 20]. We say that test case t_j contaminates test case t_k whenever t_j can alter the normal outcome of t_k if both tests are executed on separate testing machines at the same time. For example, if two test cases both perform updates against a common database and the regression testing system distributes their execution such that they are run concurrently, test case contamination could easily occur. When a distribution mechanism is used to control the execution of a regression test suite, it is still possible for one independent test to contaminate another independent test if they are executed during the same time period. Since there is no effective means to detect the possibility of test case contamination through an examination of test case source code [8], a distributed regression testing application must exploit other means to prevent test contamination. However, a regression testing process that has *a priori* knowledge of test case contamination situations should use this information to distribute the execution of tests in a fashion that effectively avoids test contamination.

The need for a load distribution mechanism is alluded to by Consideration 3. Load distribution can take the form of load balancing or load sharing [7, 28]. A load balancing approach would

attempt to ensure that each available testing machine is doing almost the same amount of work at any point in time. A load sharing approach to distributed regression testing would simply attempt to initiate the execution of a test on a lightly loaded test machine [7]. Although load sharing does not guarantee an equal workload on each testing machine, it is conceptually simpler and easier to implement. Any load distribution method must attempt to utilize the available testing resources provided by a network of heterogeneous machines in order to maximize the cost-effectiveness of the regression testing process.

Consideration 4 addresses the need for regression test suite integrity. A regression testing tool that exhibits a high level of integrity ensures the correctness of the reported results [20]. The consideration of test suite integrity arises when we distribute the execution of a regression test collection across multiple machines with different operating systems, network connectivities, and hardware configurations. For example, a temporary loss of a test machine's network connectivity should not cause a test case that would normally succeed to eventually fail. Moreover, the loss of a test machine's network connection should not force the regression testing application to omit the execution of the current test case. Addressing the consideration of test suite integrity is a challenge because we must determine if the anomalous behavior of a test case should be attributed to the test itself or to the distribution mechanism.

The topic of regression test execution control and results handling is examined in Consideration 5. The user of a regression testing tool that supports distributed test execution should be able to use the application from a central user-interface. Even though the tests are executed on multiple machines, the tester should not be responsible for extracting test suite results from the individual testing machines. Furthermore, the centralized testing environment should provide the tester with multiple methods for selecting suites for execution and browsing final test results.

5 Design and Implementation Details

In Section 3 we presented a characterization of a solution to the regression testing problem that relied upon test selection, prioritization, and distribution. Furthermore, Section 4 addressed the issues that must be considered in order to implement a reliable distribution mechanism. In this section, we examine the conceptual foundation upon which `Joshua`'s distribution approach is based and offer a high level understanding of the tool's design and implementation. The current implementation of `Joshua` directly addresses Step 3 and Step 5 in solution *RTSPD*. This version of `Joshua` contains stubs to ensure that the application can be extended with implementations of different selection and prioritization mechanisms. When a test prioritization technique is integrated with `Joshua`, it will be possible to use distributed test execution to improve the practicality of Step 2 in Algorithm *RTSPD*.

5.1 Space-Based Distributed Computing

`Joshua` relies upon an approach to distributed computing that was initially implemented in the Linda coordination language [13]. This coordination technique uses indirect communication via a tuplespace [5]. A tuplespace is a shared, network-accessible repository for objects that can be used for persistent storage and communication. The simple and elegant interface provided by a space allows loosely-coupled processes to communicate without relying upon the ability to explicitly pass messages to one another [12]. A space-based distributed computing approach relies upon the execution of a series of `write`, `read`, and `take` operations in order to communicate via a centrally located tuplespace. Refer to [5, 6, 13] for an examination for the details of space-based computation.

Our approach to distributed regression testing relies upon JavaSpaces, a Java-based implementation of the tuplespace concept. JavaSpaces allow Java programs to communicate through a shared, network-accessible "memory." Unlike standard database technology, JavaSpaces provide storage that allows for the exchange of executable content [12]. JavaSpaces are integrated with the Jini Network Technology, a set of Application Programming Interfaces (APIs) and runtime conventions that facilitate the creation and deployment of distributed systems [2, 10]. The Jini framework provides an implementation of a set of specifications that enable services to discover each other and to participate in a federation of other services that collaborate to solve a problem. For an in-depth review of Jini and JavaSpaces, refer to [2, 10, 12].

5.2 Replicated-Worker Pattern

Since each test in a test suite exhibits the same high-level behavior, the regression testing process can be decomposed into a number of smaller, independent executions of test cases. The replicated-worker distributed computing paradigm is normally applied in situations where a large problem is ideally suited for decomposition into smaller ones [12]. This approach attempts to solve a problem faster by breaking the computation into smaller units that can be executed in parallel on multiple machines. In this pattern, a master or controller divides a large problem into smaller tasks and deposits them into some implementation of a tuplespace. When a new task enters the tuplespace, one of the replicated worker extracts it, performs the appropriate computation, and writes the final results back into the space. Figure 1 provides Java-based psuedocode that illustrates the computations that would be performed by a master and replicated workers that are customized for regression testing [12].

The efficiency of regression test suite execution can be improved by the application of the replicated-worker pattern because the problem has a high computation/communication ratio. That is, the time workers spend communicating through the tuplespace with the master and the other workers is a relatively small fraction of the time that they spend computing [12]. If the tests in a regression test suite are independent [20], then none of the replicated workers should need to communicate with any of the other workers. Furthermore, there is little or no need for the workers to continue communication with the master after they have extracted a test from the tuplespace. In the unlikely situation where a substantial portion of a regression test suite includes tests that

```
public class Controller {

  for(int i = 0; i < totalTests; i++) {
   Test test = testSuite.elementAt(i);
   space.write(test);
  }
  for(int i = 0; i < totalTests; i++) {
   TestResult template = new TestResult(...);
   TestResult result =
    (TestResult)space.take(template,...);
  }

}
```

```
public class Worker {

 while(!done) {
  TestCase template = new TestCase(...);
  TestCase test =
   (TestCase)space.take(template,...);
  TestResult result = execute(test);
  space.write(result);
 }

}
```

Figure 1: The Replicated-Worker Paradigm for Distributed Test Execution.

execute very rapidly, `Joshua`'s distribution mechanism might not prove to be beneficial. However, this concern could be remedied by ensuring that the controller groups several tests together into a collection that can be executed by the replicated workers.

5.3 High-Level Architecture

`Joshua` is an extension of the widely adopted `JUnit` testing framework [16] that relies upon the Jini and JavaSpaces technologies described in Section 5.1. `JUnit` provides a test description and automation framework for Java software systems. `Joshua` focuses on the distributed execution of regression test suites that contain test cases which adhere to the `JUnit` testing framework. Whenever possible, `Joshua` reuses the existing Java classes and naming conventions that have already been established by `JUnit`. Figure 2 offers a high level description of `Joshua`'s software architecture.

When a regression tester starts `Joshua`, a centralized `TestSpace` is created on an arbitrary testing machine for the storage of regression test cases and results. During the execution of a regression test suite, the tester can rely upon the Jini-based `TestExecutors` that are currently running on accessible machines or specify that new `TestExecutors` are initialized. Furthermore, `Joshua` is responsible for the creation of a Jini-enabled version of the application under test that can be transparently accessed by each of the `TestExecutors`.[2] After the tester selects one or more regression test suites for execution, the `TestController` prepares the test cases and successively writes them into the `TestSpace`. When a given test case enters the `TestSpace`, a `TestExecutor` consumes the test and begins execution. Once the execution of the test is completed, the `TestExecutor` is respon-

[2]Due to space restrictions, we have omitted the steps that must be taken to Jini-enable a Java-based system. The process involves the conversion of all important Java objects into Jini services that be accessed through the standard Jini Discovery protocol [2, 10]

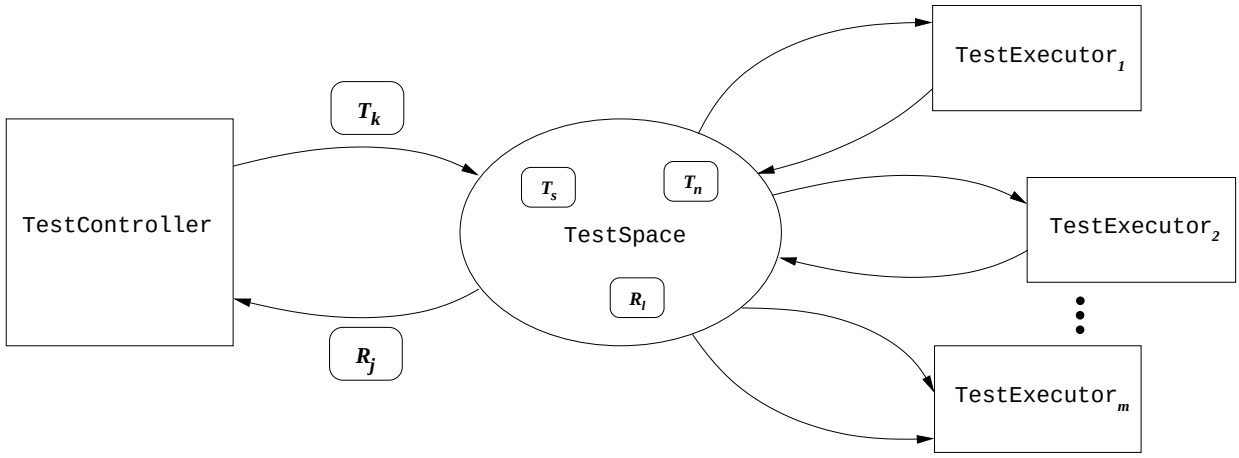

Figure 2: The High-Level Architecture of `Joshua`.

sible for preparing the results and writing them into the `TestSpace`. Finally, the `TestController` must monitor the `TestSpace` for test results and update `Joshua`'s centralized interface when they become available. The distributed regression test execution process continues until the supply of tests within the `TestSpace` has been exhausted and the results of each test execution have been reported to the tester.

5.4 Handling Distribution Considerations

`Joshua`'s existing implementation attempts to address the considerations that were enumerated in Section 4. We will now examine how `Joshua` handles each of these considerations. The Jini network technology allows for the creation of dynamic and self-healing distributed systems [10]. The current implementation of `Joshua` utilizes these fundamental features of Jini in order to ensure that the distribution process is both transparent and automatic. For example, both the `TestController` and each of the `TestExecutors` use Jini's Discovery protocol to transparently locate the `TestSpace` [2, 10]. Once a `TestExecutor` is initialized on a machine, it can automatically resume operation when new test cases are made available for execution. However, `Joshua` allows the tester to specify the names of the machines that are initially available for regression test case execution.

The current version of `Joshua` relies upon an *a priori* knowledge of test contamination possibilities that is expressed in the JavaDoc comments for a given test case. For example, the test case `testBankAccountWithdraw` might contain a JavaDoc annotation such as "`@contaminates testBankAccountDeposit`" within its comment block. This would indicate that test case contamination could arise whenever `testBankAccountWithdraw` is executed on one testing machine at the same time as `testBankAccountDeposit` is executed on another. Whenever `Joshua` is provided with a description of possible test contamination situations, it uses standard graph coloring techniques in an attempt to appropriately schedule the distributed execution of a regression suite [1, 31]. In situations where it is impossible to distribute the execution of a regression test suite in order to

eliminate all possibility of test case contamination, `Joshua` simply resorts to the linear execution of the offending portion(s) of the test suite on a single machine.

The simple distribution mechanism that `Joshua` employs ensures that the `TestExecutor` on each testing machine is always busy. A `TestExecutor` is designed to continue fetching test cases for execution until the regression test suite stored within the `TestSpace` is exhausted. Even though this approach does not exactly balance the load on each testing machine, it provides an easily implemented load sharing approach. Furthermore, any distribution mechanism that relies upon a `TestSpace` that is transparently accessible ensures that the loosely-coupled `TestExecutor`s and `TestController` can effectively communicate.

Finally, `Joshua` relies upon the transaction primitives provided by Jini in order to ensure that the results of a test case's execution are not lost or discarded if a `TestExecutor` becomes dysfunctional [2, 10]. All Jini-based transactions are governed by a transaction manager that creates transactions and monitors their operation. When a `TestExecutor` extracts a test case from the `TestSpace` under a transaction, it can complete in one of two ways. If a transaction commits successfully, then the test is removed from the space, the results are written into the space, and the `TestExecutor` is free to execute another test case. If the `TestExecutor` experiences difficulties due to any type of partial failure, the transaction is aborted by the manager and none of the commit operations occur [2, 10, 12]. The usage of a simple retry mechanism for tests that continue to remain in the `TestSpace` ensures that test integrity is preserved without wasting unneeded computational resources on non-deterministic or faulty tests.

6 Conclusions and Future Work

In this paper, we have described the conceptual foundation, design, and implementation for a tool that distributes the execution of a regression test suite. We have shown how our technique can be used in conjunction with or in isolation from other solutions to the regression testing problem. Furthermore, we have discussed the considerations that must be addressed in order to create a reliable test distribution mechanism. Finally, we have exposed the high-level implementation of `Joshua`, a tool that facilitates the distributed execution of a regression test suite for Java-based systems. We have demonstrated that `Joshua` is a flexible and extensible application that can make the regression testing process more practical and cost-effective.

The initial stages of `Joshua`'s research and development have suggested many avenues for future work. First, it is important to remove any of the artificial limitations that exist in the current implementation of `Joshua`. Next, it will be useful to integrate existing regression test selection and prioritization approaches into the framework that `Joshua` provides. The usage of `Joshua` during the regression testing of real-world software systems will certainly yield insights into further improvements. An empirical analysis that attempts to characterize the types of benefits that `Joshua` might provide for different classes of software systems is also needed. The continued evolution of `Joshua` will ensure the creation of a complete solution to the regression testing problem for Java applications.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Inc, Reading, MA, 1986.

[2] Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, and Ann Wollrath. *The Jini Specification*. Addison-Wesley, Inc., Reading, MA, 1999.

[3] T. Ball. The limit of control flow analysis for regression test selection. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 134–142. ACM Press, March 1998.

[4] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, NY, 1990.

[5] Nicholas Carriero and David Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, 21(3), September 1989.

[6] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs: A First Course*. MIT Press, Boston, MA, 1990.

[7] Thomas L. Casavant and Jon G. Khul. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2), February 1988.

[8] Martin D. Davis, Ron Sigal, and Elaine J. Weyuker. *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*. Academic Press, Boston, MA, 1994.

[9] R.A. DeMillo, R. J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[10] W. Keith Edwards. *Core Jini*. Prentice Hall PTR, Upper Saddle River, NJ, 1999.

[11] Sebastian Elbaum, Alexey G. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 102–112. ACM Press, August 2000.

[12] Eric Freemen, Susanne Hupfer, and Ken Arnold. *JavaSpaces: Principles, Patterns, and Practice*. Addison-Wesley, Inc., Reading, MA, 1999.

[13] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[14] Dick Hamlet and Joe Maybee. *The Engineering of Software*. Addison Wesley, Boston, MA, 2001.

[15] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.

[16] Ronald E. Jeffries. Extreme testing. *Software Testing and Quality Engineering*, March/April 1999.

[17] H. K. N. Leung and L. J. White. Insights into regression testing. In *Proceedings of the International Conference on Software Maintenance*, pages 60–69. IEEE Computer Society Press, October 1989.

[18] Brian Marick. When should a test be automated? In *Proceedings of the 11th International Quality Week Conference*, San Francisco, CA, May, 26–29 1998.

[19] Akira K. Onoma, Wei-Tek Tsai, Mustfa H. Poonawala, and Hiroshi Suganuma. Regression testing in an industrial environment. *Communications of the ACM*, 41(5):81 – 86, May 1998.

[20] Bret Pettichord. Seven steps to test automation success. In *Proceedings of the International Conference on Software Testing, Analysis, and Review*, San Jose, CA, November 1999.

[21] G. Rothermel and M. J. Harrold. A framework for evaluating regression test selection techniques. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 201–210. IEEE Computer Society Press, May 1994.

[22] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, August 1996.

[23] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.

[24] G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering*, 24(6):401–419, June 1998.

[25] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In *Proceedings of the International Conference on Software Maintenance*, pages 179–188, August 1999.

[26] G. Rothermel, Roland H. Untch, Chengyun Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*. To Appear.

[27] Stephen R. Schach. *Software Engineering*. Aksen Associates, Boston, MA, 1992.

[28] N. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *IEEE Computer*, 25(12):33–44, 1992.

[29] F. Vokolos and P. Frankl. Pythia: A regression test selection tool based on textual differencing. In *Third International Conference of Reliability, Quality, and Safety of Software Intensive Systems*, May 1997.

[30] James A. Whittaker and Jeffrey Voas. Toward a more reliable theory of software reliability. *IEEE Computer*, 32(12):36–42, December 2000.

[31] Robin J. Wilson. *Introduction to Graph Theory*. Addison-Wesley Longman Limited, Edinburgh Gate, England, 1996.

[32] W.E. Wong, J.R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proceedings of the Eight International Symposium on Software Reliability Engineering*, pages 230–238, November 1997.