City University of Hong Kong
Department of Computer Science

BSCCS Final Year Project 2013-2014

13CS009

RegressionMaple: Regression coverage of concurrent testing on validating bug-fixing

(Volume 1 of 1)

Student Name : **Tsui To**

Programme Code : **BScCS**

For Official Use Only

Supervisor : **Dr. CHAN, Wing Kwong**

1st Reader : **Dr. WANG, Jiying**

2nd Reader : **Dr. YU, Yueng Tak**

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to Dr. Wing-Kwong Chan, my supervisor of final year project, for his patience, guidance, enthusiastic encouragement, and valuable advices. His nurturing helped me in all time of this project and extended to my growth in nonacademic aspects.

TABLE OF CONTENTS

RegressionMaple: Regression coverage of concurrent testing on validating bug-fixing

Tsui To

1 ABSTRACT

Multicore hardware makes performance faster. With the pervasiveness of software and hardware support, concurrent computing is widely applied. While enjoying its benefits, there is also a new challenge — concurrency bug. Concurrency bug is an error caused by incorrect thread interleavings. In concurrent computing, threads are interleaved with each other to simulate as executing in parallel. But, in fact, threads are executed one by one in a small time slice, and communicate with each other (for example, via shared memory). Maple is one of several software of automatic concurrency bugs detection, successfully applying dynamic analysis to reveal concurrency bugs such as data race and deadlock. In addition, it generates histories of tested and failed-to-test interleaving schedules. It gives a progressive method for developers to test their concurrent software. While Maple is good at detecting concurrency bugs with respect to the same input, it is not without its flaws. This project has observed two situations, in which Maple is possible to be improved. The two situations are lack of accurate coverage across versions and inability of validation on concurrency bug-fixing. First, Maple treats versions of a program as totally different programs. It requires a full set of retest processes on every version. It is clearly a time consuming process, as developers and testers are often under stress to release a new version. Second, once a concurrency bug is exposed by Maple, developers will try to resolve it. However, after suspicious codes were modified, the developers have no information to determine if the concurrency bug is completely fixed or not. In the current approach, they can only retest the possible interleaving schedules but without any target in mind. To this end, this project proposes a new regression coverage driven testing tool — RegressionMaple. It applies the concept of regression testing (with assumption of similar execution context) to link testing information across two versions of a program, thus improves Maple with respect to the above two problems.

2 MOTIVATION AND BACKGROUND INFORMATION

Concurrency bug is an error caused by incorrect thread interleavings. In concurrent computing, threads are interleaved with each other to simulate as executing in parallel. But, in fact, threads are executed one by one in a small time slice, and communicate with each other (for example, via shared memory). It is unpredictable on which thread schedule to be actually executed in production. If an instruction executed by a thread overwrites the value of a shared variable that has been written by another thread and supposedly it should be unchanged, then a concurrency bug (e.g., data race) occurs. Because of unpredictable schedules in production, the occurrence condition of a concurrency bug is not guaranteed to being satisfied in every run. Concurrency bugs are difficult to reveal, reproduce and resolve. Stress testing [1] is one of several methods widely used in the industry. It repeatedly executes a program in extreme environments with the belief of being able to achieve a high coverage of thread interleavings thus discover these buggy ones. On the other hand, active testing has proven to be efficient in exposing concurrency bugs by monitoring actual interleavings at runtime. Maple [2] is one of several tools of concurrent testing that successfully apply

dynamic analysis to reveal bugs such as data races and deadlocks. In addition, Maple generates histories of tested and failed-to-test interleaving schedules. It gives a progressive method for developers to test their software. By controlling interleaving schedules and according its testing histories, each test run for the same input can potentially expose new concurrency bugs more efficiently and effectively than stress testing.

3 PROBLEMS

Although Maple is good at detecting concurrency bugs with respect to the same test input, it is not without its flaws. This project has observed two situations that this project can possibly improve Maple. The two situations are lack of accurate coverage across versions and inability of validation on concurrency bug-fixing.

3.1 Lack of accurate coverage across versions

It is infeasible to apply Maple to software engineering especially in the aspect of coordination of different versions of a program. Maple treats each version of same software as an individual one. It requires a complete retest on all affected inputs to ensure an acceptable level of thread interleaving coverage, even if there are only minor changes in the code. We may imagine that there can be a large volume of possible thread interleavings with respect to each input. This approach fundamentally incurs a very large state space exploration. Furthermore, accurately examining inputs which are affected is also an expensive task. Retesting the whole program is clearly time consuming, because it should only target the modified parts and potentially affected parts. Although it is not easy to determine those parts without false negatives, achieving this can save plenty of time cost on regression testing.

3.2 Inability of validation on concurrency bug-fixing

Maple lacks the ability to validate whether a concurrency bug is fixed. In functional testing, the validation can easily be carried out by comparing the expected output with the actual output. In concurrent testing, such validation is inapplicable. Instead, such a testing process depends on active scheduling of instructions to trigger concurrency bugs. After the repair of a concurrency bug, the instructions of the modified program are different from the original one. If the schedule had not been modified, then the previous buggy parts can be reexamined. However, this is not the general case. Since the instructions are modified during bug correction, the schedule may become different. Therefore, it is no longer guaranteed that the previously buggy parts can be reexamined. In short, this difference causes bug reproduction approaches (such as that of Maple) unable to assess the effectiveness of a concurrency bug correction.

4 PROJECT OBJECTIVES AND SCOPE

This project proposes a new regression coverage driven testing tool — RegressionMaple. It applies the concept of regression testing [3] to improve Maple with respect to the above two problems, namely, lack of accurate coverage across versions and inability of validation on concurrency bug-fixing.

4.1 Regression coverage

This project introduces a method called regression coverage to coordinate different versions of the same program. It will extract two sets of data from the original version: one is the testing information, and the other is a set of determinants. The set of determinants is used

to assess the usability of the testing information across different versions. Based on the set of determinants, the useful parts of the testing information will be ported from the original version to a newer version of the same program. The testing information should provide: (1) data on the unchanged parts that can be injected into the new version; and (2) partial data about interfacing the unchanged parts to changed parts. With regard to the design of Maple, it is possible that porting usable histories to a newer version may cut the time needed for retesting the stable parts and in addition can expose untested thread interleavings based on the background information.

4.2 Validation on concurrency bug-fixing

This project provides an interface for assessing corrective actions (e.g. update patch) on previously exposed bugs. The bugs detected can be reproduced by providing the interleaving schedule to Maple. This project makes use of this function by transferring the schedule across versions. The transferring process depends on the information collected from the original version, how these pieces of information compare to the newer version. Differentiating them can provide a small set of untested interleavings for the connections between the unchanged parts and the changed parts. The bugs of original version can then provide information to test the most suspicious interleavings with respect to the corrective actions. As a result, it can provide a higher level of confidence that the bug was fixed.

5 LITERATURE REVIEW

To comprehend and exploit concurrent testing and its principles, this section covered (1) the major alternatives: *stress testing*, *systematic testing*, and *active testing*; and (2) state of the art of the techniques and knowledge in this domain. They are *concurrency bugs*, *coverage and detection techniques*, *regression testing*, and *source-code differentiation*.

5.1 Major alternatives

Stress testing, *systematic testing* and *active testing* are the major methods of concurrent testing (i.e. the testing to reveal concurrency bugs). Their main ideas to test concurrent software are different. They have different advantages and disadvantages. In this section, it will discuss these three approaches and compare them with this project.

5.1.1 Stress Testing

Stress testing [1] is widely used in the industry to test concurrent software. It repeatedly executes the software in heavy load situations. The idea of it is to simulate such extreme situations to cover more thread interleavings. A *thread interleaving* occurs when a thread execution is interrupted by another thread. Every interleaving may potentially expose a concurrency bug. Thus, the idea of revealing concurrency bugs by increasing interleaving coverage is making sense. But there are two uncertainties: (1) repeatedly executing concurrent software is not effective to reveal new interleaving [2]; and (2) *stress testing* intuitively cannot reproduce revealed bugs [4].

To this end, a recent work [4] has verified these two doubts that *stress testing* is both ineffective and not reproducible. The work carried empirical studies to evaluate the effectiveness of stress testing in regard to the ability of covering more interleavings. It found that interleavings hold individual occurrence probabilities, and the range of them can be from almost 0% to almost 100%. Thus, some of those interleavings can never be exposed even after a long time execution.

Although *stress testing* is not effective to reveal concurrency bugs, it provides a basic idea to study the characteristic of concurrency bugs (e.g. the studies of [4]). Our project adopts the use of interleaving occurrence probability as the basis. It supports the idea of this project which applies *active testing* to actively control interleavings schedule instead of adopting *stress testing*.

5.1.2 Systematic Testing (Model Checking)

Systematic testing [5] (also referred as *model checking* or *state-space exploration*) is an effective method to guarantee the absence of concurrency bugs in concurrent software. Systematic testing mainly consists of *state-space* and *state-space property*. First, the testing will analyze the state-space of the concurrent software. *State-space* can be viewed as a tree that shows every combination of the system behavior. Then, the testing will verify every behavior by *state-space properties* (e.g. deadlocks, atomicity violations, data races).

A major drawback of systematic testing is scalability problem. Because the state-space can be extremely large, systematic testing can take a long time to verify large scale software, and the real software is usually large scale. Some recent works [6] [7] have improved this situation. Partial-order reduction [6] and context-switch bounding [7] can effectively reduce the state-space to explore.

However, even with these techniques improved the situation, the space to explore for real software is still large. This problem of huge exploration space makes the application of systematic testing impractical for the industry because software often faces the problem of urgent-fixing updates. Despite of this disadvantage, the ability of guaranteeing correctness of concurrent software can improve life-critical system (e.g. energy system [8]).

5.1.3 Active Testing

Active testing [2] [4] [9] [10] [11] is an effective and efficient concurrent testing method. It consists of two phrases: (1) reveal suspected thread interleaving schedules which may potentially trigger concurrency bugs; and (2) execute each discovered schedule to validate and expose concurrency bugs. A critical concern of active testing is the efficiency and effectiveness of which techniques apply in bug detection (i.e. the process of phrase 1). Concurrency bug detectors act as an important role in active testing.

There are several recent works in different areas of active testing especially in bug detection. Maple [2] introduced a set of bug patterns to simplify bug detention complexity, designed a memorization technique to boost thread interleavings coverage, and in addition, it adopted a technique to avoid deadlock occurrence during validation process (i.e. the phrase 2). Ctrigger [4] defined a set of simple atomicity violation patterns that can effectively expose suspicious interleavings. RaceFuzzer [9] designed an approach to realize given thread interleavings at runtime.

The state of the art of active testing techniques successfully reveals concurrence bugs. Nonetheless, it lacks ability to validate whether a concurrency bug is fixed since the interleaving dependences may already change after program modification. In addition, it does not consider the reality of software evolution. Because of these two issues, it obstructs the application in the industry. This project will focus on solving these two problems to facilitate the industrial adoption on active testing techniques.

5.2 Current status and limitation

Concurrent testing is an active field in research. In the state of the art, there are some novel and insight ideas, techniques, principles, and studies proposed. In this section, it will cover the current status and limitation to the major areas related to concurrent testing.

5.2.1 Concurrency Bugs

Concurrency bug is danger to the world and needed to be eliminated. According to a news in 2004 [8], there is a serious blackout in northeastern U.S. due to a concurrency bug in a life-critical system (i.e. an energy management system).

Concurrency bug is an error caused by thread interleavings. It may lead to incorrect results after a success execution. In concurrent computing, threads are interrupted (i.e. thread interleavings) with each other to simulate as executing in parallel. But, in fact, threads are executed one by one in a small time slice, and communicate with each other (for example, by shared memory). It is unpredictable that which thread schedule will actually be executed in production. If an instruction in a thread overwrites the value of a shared variable that has been written by another and supposedly is unchanged, then a concurrency bug (i.e. data race) occurs. Because of unpredictable schedule, concurrency bug occurrence is not guaranteed to be satisfied in every run.

Concurrency bugs are difficult to reveal. They can be classified into a few major categories: *data races*, *atomicity violations* and *deadlocks*. Concurrent testing which aims at validating absence of concurrency bugs in software is actively in research. Several recent works [2] [4] [12] [13] have defined different concurrency bug patterns to detect suspicious interleavings which are potentially to manifest a concurrency bug.

Data race is a type of general concurrency bugs. If a result of execution running concurrently can be different to running sequentially, then a data race occurs [4]. In general, *data race* refers to such a concurrency bug that involves only *one* inter-threads shared variable.

Atomicity violation is a common concurrency bug. It is an error that the developer made a wrong assumption of which the read or write action in shared variable is safe. By being safe, we mean that the actions are executed atomically (i.e. no interruption is allowed in such block of actions) [9].

Deadlock is a problem of infinite blocking where the affected threads infinitely wait for a release of shared variable of another waiting thread. It occurs due to the nature of synchronization.

5.2.2 Coverage and Detection Techniques

Interleaving coverage [2] is a technique to assess the quality of concurrent testing especially in *active testing*. Similar to the typical coverage techniques (e.g. code coverage) in sequential program testing, it can provide an assessment to testers to help them to decide when to stop and how the quality is. A main idea of *interleaving coverage* is that it assesses the testing quality by calculating the quantity of examined thread interleaving in the set of all feasible suspected thread interleavings. Several recent works [2] [4] [14] [11] have proposed different techniques to determent. They are:

- Maple's set of idioms [2]
- CTrigger's atomicity violation patterns [4]
- Concurrent Function Pair [14]
- PENELOPE's algorithm of atomicity violation schedules detection [11]

Maple [2] has introduced six interleaving patterns (i.e. refer to as idioms) that are simple and effective. This patterns covers single-variable dependences and double-variable dependences (i.e. atomicity violation). The empirical results show that the patterns can efficiently expose the suspicious buggy thread schedules without significantly dropping down effectiveness.

Ctrigger [4] has proposed four atomicity violation patterns that can effectively expose hard-to-detect bugs. Those bugs usually hold a very low occurrence probability in normal case.

Concurrent Function Pair [14], in the input-level, coordinates a set of inputs to filter out such redundant thread interleavings across inputs. It significantly drops the test time by predicting each input's contribution to the whole set of interleavings.

5.2.3 Regression Testing

Regression testing [15] is a testing activity which often performs after each change of a program so that confirms the changed parts of the program do not introduce new errors in the unchanged parts. It is crucial to software quality. Typically, regression testing retests the existing test suites which continuously enlarge during software evolution. Because of the time constraints, it seems impossible to re-execute the whole test suite in real software.

Therefore, several techniques [3] have been proposed to deal with time constraint problem. The techniques include *retest all*, *test case prioritization* and *test case selection*.

Retest all [3] requires executing all existing test cases. Although this technique consumes plenty of time, it can guarantee that no new errors introduced in the unchanged codes.

Prioritization [3] consists of two phrases: (1) predict the priority of each test case with regard to the changes; (2) run the prioritized test cases.

Selection [3] aims at selecting the most valuable test cases to be retested. First, it will predict the cost of *Retest all* and the cost of pruning process. It will choose the approach by comparing the cost of *retest all* and the cost of examining test cases to prune. The pruning can be based on three criteria: (1) *coverage* which only retests the parts modified and other parts may bring influenced; (2) *minimization* which is a kind of *coverage*, but only execute the smallest set of test cases; (3) *Safety* which, instead of employing *coverage*, investigates the output differences to select test cases. Safety selection measurement can also divide into *inclusiveness*, *precision*, *efficiency* and *generality*.

6 PROPOSED DESIGN, SOLUTION, SYSTEM

This project proposes a tool — RegressionMaple. It exploits idea of regression testing in concurrent testing. To the best of knowledge, this idea is novel and therefore is challengeable. In this section, it will discuss the design of the system to deal with the challenges in this project.

6.1 Reviewing the design of Maple

This project is heavily depending on Maple [2]. Therefore, this section discusses Maple's main idea and components which are relevant to this project. The general design of Maple is same as typical dynamic analysis tools. It consists of two phases. They are profiling phase and scheduling phase. Profiling phase accepts a test input and then predicts all feasible schedules of this input. Scheduling phase will then examine each predicted schedule to verify its feasibility. Maple defined coverage of interleaving instructions and aims at discovering as many feasible schedules as possible. By exploiting this idea, Maple is effective in the exposure of concurrency bugs.

6.1.1 iRoot – interleaving instructions schedule

A schedule is not a fully strict schedule. Instead, it is a partial strict schedule. Maple refers each schedule as an *iRoot*. An iRoot manipulates a sequence of two to four instructions. It remains other unmanaged instructions executed arbitrarily. This approach is effective and retains efficiency because only a small set of instructions is related to a concurrency bug.

6.1.2 Idiom – buggy interleaving pattern

Maple generalized the patterns of two-thread based schedule. Six types of iRoot are shown in Figure 6.1. They are idiom 1 to idiom 6. Idiom 1 consists of 2 events. Each event is an instruction which either accesses a shared memory location or accesses a lock. Each idiom is a pattern of happen-before relationship of its events. For example, a read-after-write iRoot, which belongs to idiom 1, is a two-event iRoot where a thread read a memory location and then another thread writes the same memory location.

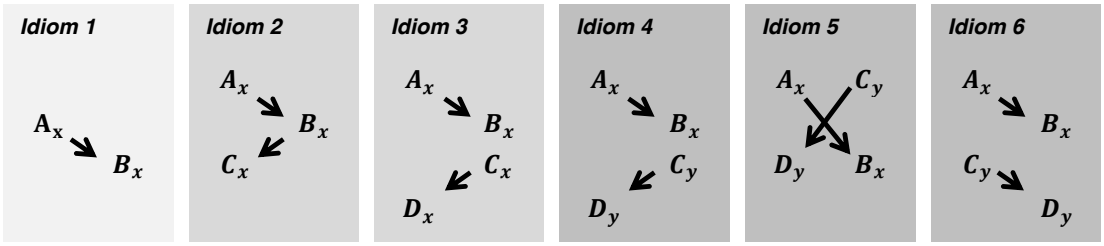


Figure 6.1 - Six idioms for two threads (taken from [2])

To further explain, as shown in Figure 6.1, A_x is an instruction A which accesses a shared memory x. B_x, C_x and D_x are other instructions accessing a shared memory x. The idiom 1 is a simple idiom, while idioms 2 to 6 are complex idioms. Maple looks for every idiom 1 and memorizes them. Complex idioms are recognized by utilizing a *vulnerability window (vw)*. *Vw* is an amount of instructions allowed to be executed between two events defined in each complex idiom. For example, given *vw = 1000* and considering a idiom 2 pattern, if two iRoots of idiom 1, namely $iRoot_i$ and $iRoot_j$, are exposed in different threads, and they are relevant that $iRoot_i = A_x \; then \; B_x$ and $iRoot_j = B_x \; then \; C_x$, Maple will predict a

$iRoot_{(i,j)} = Ax, Bx, Cx$ in type of idiom 3. The vulnerability window *vw* is an additional requirement must be stratified to predict that iRoot. It is the number of instructions allowable before the exposure of a complex idiom. Considering the example again, if the number of instructions executed between the exposure of $iRoot_i$ and $iRoot_j$ exceeds 1000 (i.e. the vw) the $iRoot_{(i,j)}$ will not be predicted.

6.1.3 Profiling phase – predicting iRoot candidates

In the profile phase, Maple makes use of Probability Combination Test profiler to provide a random schedule of interleaving instructions. Maple then observes the exposed iRoots and predicts feasible iRoots.

Profiler is a type of `execution-control`. It is an online controller to manipulate the instructions execution at runtime. Profiler instruments every suspected instruction, which is a memory access instruction, and tries to discovery as many memory accesses as possible. The discovering process is delegated to two `analyzers`, namely observer and predictor.

Before and after every execution of suspected instructions, an instrumentation tool, i.e. Intel's PIN tool, will notify the profiler and it will invoke the observer and the predictor. Observer and predictor will record run-time memory access history for difference purposes. Observer looks for actually exposed iRoots during runtime while predictor looks for all possible combinations among them.

6.1.4 Scheduling phase – examining iRoot candidates

In the active phase, Maple examines every predicted iRoot to verify their feasibility, i.e. whether the inter-thread interleaving instructions schedule can be executed at runtime. This is also done by an `execution-control` and an `analyzer`.

Active scheduler is a type of `execution-control` combined with the role of `analyzer`. In each active run, it targets for an iRoot and tries to execute the program according to the predicted order of iRoot events. The scheduler will instrument the instructions which belong any events of the iRoot or which are memory accesses. The former offers iRoot execution requirements and the later offers run-time memory accesses information. Based on the requirements and the run-time information, the scheduler can actively control the runtime thread schedule to ensure that an iRoot is either exposed or infeasible.

Before an iRoot event's instruction being executed, an instrumentation tool, i.e. Intel PIN, will notify the scheduler and it will record its current iRoot event. Before a memory access instruction being executed, PIN will notify the scheduler and it will decide whether postpones the instruction execution depending on the iRoot predicted schedule. If an instruction accesses a memory location which is overlapped with, however the instruction is not a member of the currently targeted iRoot event, the instruction will be postponed. The targeted iRoot event's instruction will eventually be executed, otherwise the iRoot is considered as unfeasible and the process will give up to this iRoot. The successfully executed iRoot's event will then be recorded as current event. This whole process will continue until all events belonging to the iRoot are successfully executed, or will terminated in case of one of the events is infeasible.

6.2 Major technical components

Maple has two major components, namely, `execution-control` and `analyzer`. Figure 6.2 shows an abstract presentation of the design of Maple. `Execution-control` will actively control the schedule of thread interleavings. `Analyzer` will collect the active data of current test run.

Figure 6.3 shows the high-level design of this project. This project proposes four new components, building on top of Maple to implement our own solution. The components are `collector`, `converter`, `differentiator` and `manipulator`.

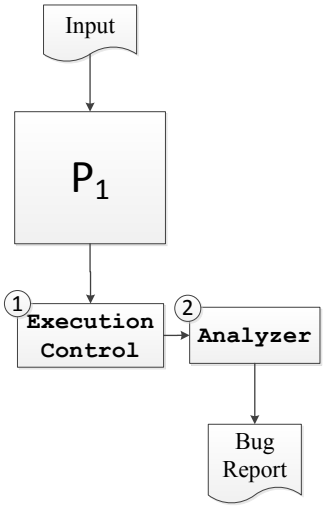


Figure 6.2 - Major components of Maple

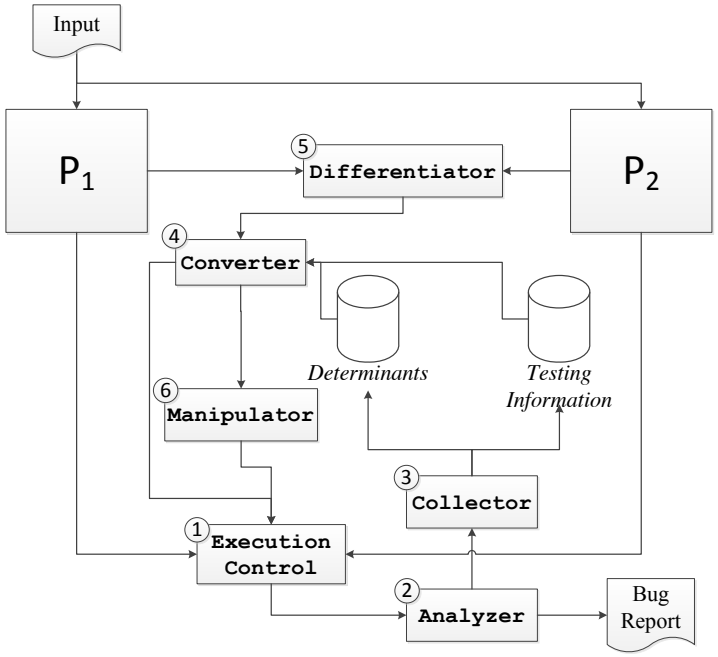


Figure 6.3 - Major components of this project

`Collector` will collect two sets of data from the original version (i.e. P_1): one is the *testing information*; another is a set of *determinants*. The set of *determinants* is used to assess the usability of the *testing information* across different versions. Based on the set of *determinants*, the useful parts of the *testing information* should be possible to port from the original version to the newer version of the same program.

`Converter` will transfer the *testing information* from the original version to the newer version based on the set of *determinants* and the differentiation results generated by `differentiator`. The converted *testing information* should provide: (1) data on the unchanged parts that can be injected into the new version; and (2) partial data about interfacing the unchanged parts to changed parts. The data on the unchanged parts will then bring injected into the testing database of the new version. `Profiler` can profile the newer version (i.e. P_2) according to the injected data.

`Differentiator` will provide extra information about the differences of the program structure between two program versions by comparing the differences between the original

version and newer version. Then generate a differentiation reports for the converter to transfer the *testing information*.

`Manipulator` will process the partial data generated by `converter`. The partial data can provide information to interface the unchanged parts to changed parts. `Manipulator` can then suggest most suspicious interleavings schedule to the profiler based on the bug in original version which is expected to be correct in the newer version. In short, `manipulator` can certainly assess the bug corrective actions.

6.3 Technical challenge in the components

There are two major challenges in this project: (1) *Regression Coverage* (i.e. the coverage approach defined in this project) is difficult to measure. The main idea of *Regression Coverage* is to provide interleavings coverage across versions. It consists of comparisons between multiple versions of software and abstractions of interleavings; (2) *Concurrency Bug-fixing Validation* is also a challenge. In order to validate whether a concurrency bug is fixed, reexamining the thread schedule in the new version of software is required. This quite makes sense because if the bug disappears in the new version in the reexamination, it can confirm that the bug is fixed. However, it is almost impossible to assure that the reexamination in a new version is same as the examination in its original version. In fact, exactly the same examination is often impossible to reproduce due to source code changes (i.e. the problem of changes of the interleaving instructions). Thus, reexamination is almost impossible in software evolution. To deal with these challenges, this project decomposes them into smaller pieces:

- *Thread schedule abstraction*
- *Regression data collection* addressed by `collector`
- *Regression comparison* addressed by `differentiator`
- *Regression interfacing* addressed by `converter`
- *Regression manipulation* addressed by `manipulator`

Thread schedule abstraction is the problem of abstracting the interleaving instructions of two software versions (i.e. an original version and a new version) in different abstraction levels to match up, and finally producing a fairly possible schedule in the new version. In software evolution, source codes change frequently. This can be due to several reasons such as code refactoring, bug correction and functional enhancement. These modification activities can proportionally affect an original schedule (i.e. the bug-triggering schedule in original version of software).

Regression data collection is the problem of collecting *significant* testing information in efficient and effective manners. The information must be able to represent the corresponding thread schedule, however, fairly remains its state across the software evolution. Each memory access or lock acquisition may need to collect those data. The process of collecting data will be heavily involved. Thus, the performance is important. In other word, the data collected must be lightweight, however, enough to facilitate the later *regression interfacing*.

Regression comparison is the problem of comparing two software versions (i.e. an original version and a new version) to support *interfacing* between them in *thread schedule abstraction*. This comparison result is treated as supplementary of thread schedule to build a

relatively strong connection across versions. *Regression data collection* aims at collects *thread schedule abstraction* data. Even though the schedule data is extracted in an abstract form, those data can be individual to its own program version. Although the abstract form gives a relationship between versions, it can be weak in a case that considerable design changes introduced in a new version. By exploiting this supplementary, the *regression interfacing* process can relate the design changes with the abstract thread schedule. This connection will be able to provide stronger support to the interfaced schedule.

Regression interfacing is the problem of interfacing two software versions (i.e. an original version and new version). This process, in the new version, selects a most relevant thread schedule by the mean that the abstract thread schedules are closet in both new and old versions. Thread schedule data are mainly depends on memory accesses and lock acquisition. Thus, the process will be involved heavily to determine the best matching schedule. An efficient algorithm to solve this problem will be needed.

Regression manipulation is the problem that actively schedules the thread interleaving in the new version according to the interfaced *thread schedule abstraction*. This is a typical online decision making problem. Under a partially executed schedule, for each monitored instruction, the manipulation process must decide whether allow or postpone the execution before the provision of its best matching schedule. However, this decided online execution schedule must reflect the best matching schedule, which provide by the interfacing process. So, this online decision is a kind of prediction that exploits the partial result provided by interfacing process to execute a fairly matched schedule.

6.4 Use-case modeling

RegressionMaple is designed to enhance the original Maple tool so that improve the two situations described in section 3. This section will discuss the use cases of RegressionMaple.

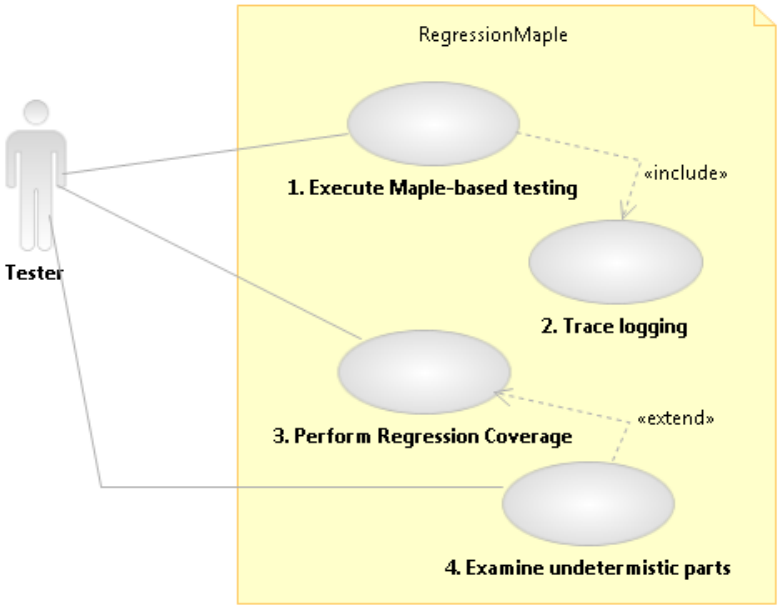


Figure 6.4 - Preliminary use-case diagram

As shown in Figure 6.4, RegressionMaple consists of 4 use cases, namely, (UC1) execute maple-based testing, (UC2) trace logging, (UC3) perform regression coverage, and (UC4) examine un-deterministic parts. Each of them focuses at solving specific problems and meeting individual purpose. They can be separately invoked and coordinate together as a process.

In general, UC1 to UC4 will be involved sequentially. Consider a multithreaded program P with its versions p_1 and p_2. After p_1 is developed and ready for test, (UC1) the tester executes maple-based testing on p_1. In this process, RegressionMaple generates a set of predicted interleaving instructions schedule $S = \{s_1, s_2, \ldots s_n\}$ and schedules each execution of S. Every feasible schedules of p_1 will be examined in this state. (UC2) RegressionMaple in addition records every result of interleaving instructions schedule, i.e. a feasible schedule or infeasible schedule. In the examination of feasible schedules, a functional test case may reports assertion. The tester then record the corresponding schedule, say s_i. The development team examines the concurrency bug and produces p_2 which tries to fix the bug exposed by s_i. (UC3) Tester tries to reexamine p_2 by setting schedule s_i and provides the testing information of p_1. RegressionMaple selects the corresponding schedule s'_i from all possible schedules $S' = \{s'_1, s'_2, \ldots s'_n\}$ and examines s'_i. The functional test case may also reports assertion. This indicates that the bug fixing is not working. Otherwise, the tester can decides whether the bug is fixed according to the confidence value which is a similarity between s_i and s'_i. (UC4) The remaining un-deterministic schedule $S' - s'_i$ can then be scheduled to examine them.

6.5 System architecture

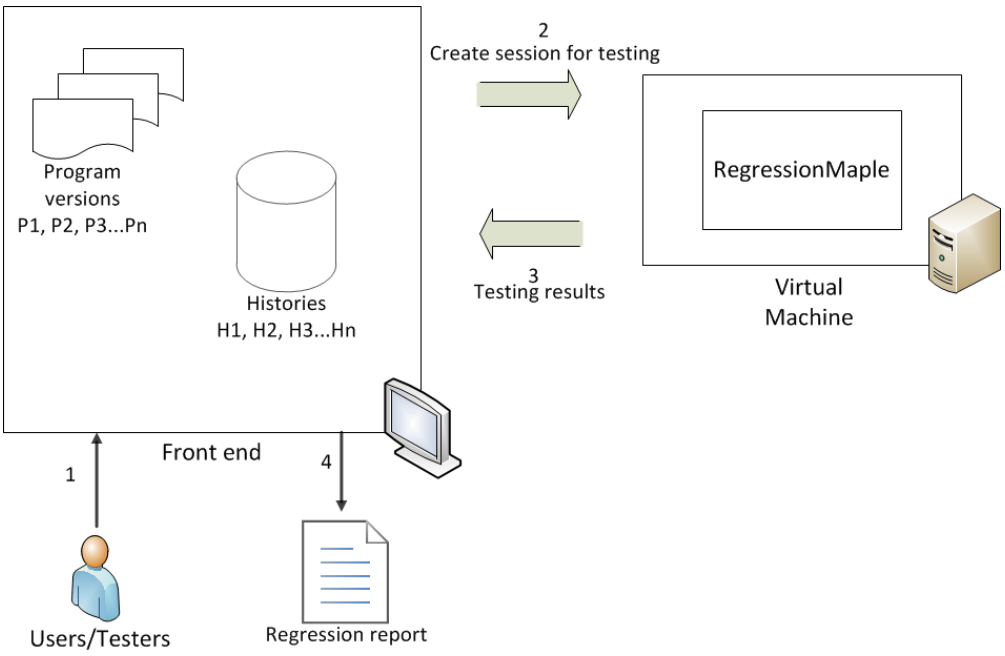


Figure 6.5: System architecture

The system consists of two major parts: (1) `front-end` which provides user interface for users operations; (2) `virtual machine` which set up a RegressionMaple within it and process concurrent testing coordinating with `front-end`.

`Front-end` contains *program versions* and their *testing histories* (i.e. the results generated by RegressionMaple). *Program versions* denoted in P_1, P_2, P_3 to P_n. *Program histories* denoted in H_1, H_2, H_3 to H_n. For example, program P_1 is consistent with history H_1. Every time to process a concurrent testing, the `front-end` will send the request (which contains the `program versions` to test and the consistent `histories`) to the `virtual machine`, and the RegressionMaple resided in the `virtual machine` will then send back results to `front-end`.

Virtual machine contains a RegressionMaple. The virtual machine virtualizes the complex underlying requirements (e.g. Ubuntu 12.04 OS, PIN [16] library and a list of package dependencies). It provides a portable approach to simplify the complex setup of this system.

6.6 Hardware and Software

This project proposes a tool called RegressionMaple which is an enhancement of an existing tool called Maple [2]. RegressionMaple is developed in Linux Perform. In this project, Ubuntu 12.04LTS is applied. C++, Bash shell script, Python and Make are the major programing and scripting languages applied in this project for implementing RegressionMaple. The hardware is supported by a virtual machine which provide a dual-core CPU which is Intel Xeon X5560 @ 2.8GHz, 2 GB RAM, and 64-bits architecture environment.

7 DETAILED METHODOLOGY AND IMPLEMENTATION

A major challenge in RegressionMaple is to deal with the schedule conversion from an old version of a program to its new version. A program context can help to deal with this problem. In this section, it will discuss the methods to obtain the execution context as well as the tradeoffs of different approaches.

7.1 Thread schedule abstraction

A major challenge in this project is *thread schedule abstraction*. In regression coverage, it is crucial to abstract the thread interleavings in certain levels so that can assess a replacement (i.e. changed from the view of the original version) interleaving in the new version. Such abstraction levels should be dynamic to provide flexible of accurate replacements in the new version. Consider an object-oriented program consist of `packages`, `classes`, `methods`, `statements` and `instructions`. They are the level of context in a program. The algorithms this project proposed is mainly depending on this nature of program context.

Abstracting program context (as shown in Figure 7.1) can provide certain level of comparison between two software versions (i.e. an original version and a new version). In the abstraction, a package P consists of a list of classes C_1, C_2, C_3 to C_n, in each of class, say C_1, consists of a list of methods M_1, M_2, M_3 to M_n, in each of method, say M_1, consists of a set of statements S_1, S_2, S_3 to S_n, and each statement is directly consistent with an instruction, for example, S_1 is consistent with an instruction I_1 (in C or C++ programming language).

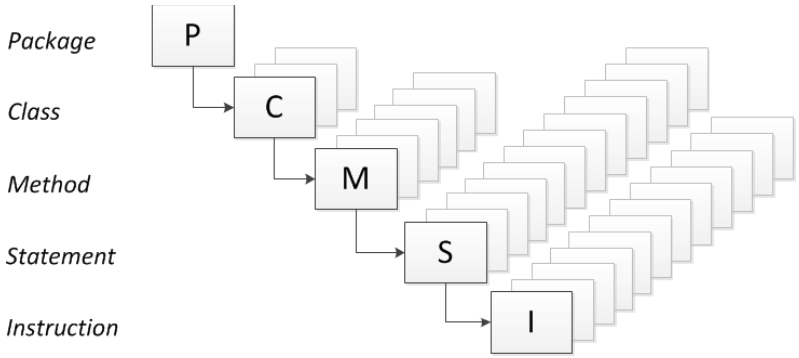


Figure 7.1 - Program context abstraction

Thread	Statements executed in thread		
T1	`function exe_remove(): remove() log_remove()`	`function remove(): lock(L1) clear content unlock(L1)`	`function log_remove(): lock(L2) log[] = "remove" unlock(L2)`
T2	`function exe_add(): add(E) log_add(E)`	`function add(e): lock(L1) content[] = e unlock(L1)`	`function log_add(e): lock(L2) log[] = "add e" unlock(L2)`

Figure 7.2: An example of atomicity violation

	T1: exe_remove()	T2: exe_add()
1	remove()	
2		add(E)
3	log_remove()	
4		log_add(E)

Figure 7.3: A bug-triggering thread schedule

An atomicity violation manifests when a bug-triggering thread schedule examined (as to shown in Figure 7.2 and Figure 7.3). In the example, the developer made a wrong assumption that `remove()` and `log_remove()` will be executed atomically (or on the other hand, `add()` and `log_add()`). This bug-trigging schedule can be represented in different levels: statement level (i.e. instrument level), and method level. Both levels can serve *same effectiveness* with regard to bug-triggering.

With respect to the *same effectiveness*, there is an extra advantage to abstract an interleaving schedule in method level instead of statement level. It is flexibility. In software evolution, source codes change frequently, but the architecture. The abstraction level described above is a kind of context inherited from the nature of software architecture. Although source code modification sometime is due to refactoring (in this case, it will drop the effectiveness), an idea that monitors the shared-variable to produce thread schedule can fill the room. Especially, in concurrency bug-fixing validation, abstraction level thread schedule can make great contribution. It can provide certain confident to reexamine the interleavings which changed in the new version.

7.2 Collector - execution context collection

Call stack is a presentation of execution context. A typical call stack mainly depends on two CPU registers. They are stack frame pointer and stack top pointer. For Intel processor, a stack frame pointer can be a register of BP, EBP or RBP, and a stack top pointer can be a register of SP, ESP or ESP, depending on the architecture of 16-bits, 32-bits or 64-bits, respectively. For example, in Figure 7.4, it shows a call stack in which a method `m1()` has called `m2()` and `m1()` is waiting for `m2()` to returns control.

In a call stack, a frame pointer (e.g. EBP) is pointing to an address of call stack which stored the caller's frame pointer address. By backtracking (i.e. dereferencing EBP), an execution context can be retrieved with low overhead. In multithreading, each thread has an individual call stack and its part of call stack will be activated when the thread is executing. In Algorithm 1, it unwinds a call stack to provide abstraction of execution context; `ebp` is the frame pointer; `thd_ebp` is the base frame pointer of the targeted thread; and `st` is the return of abstraction of execution context.

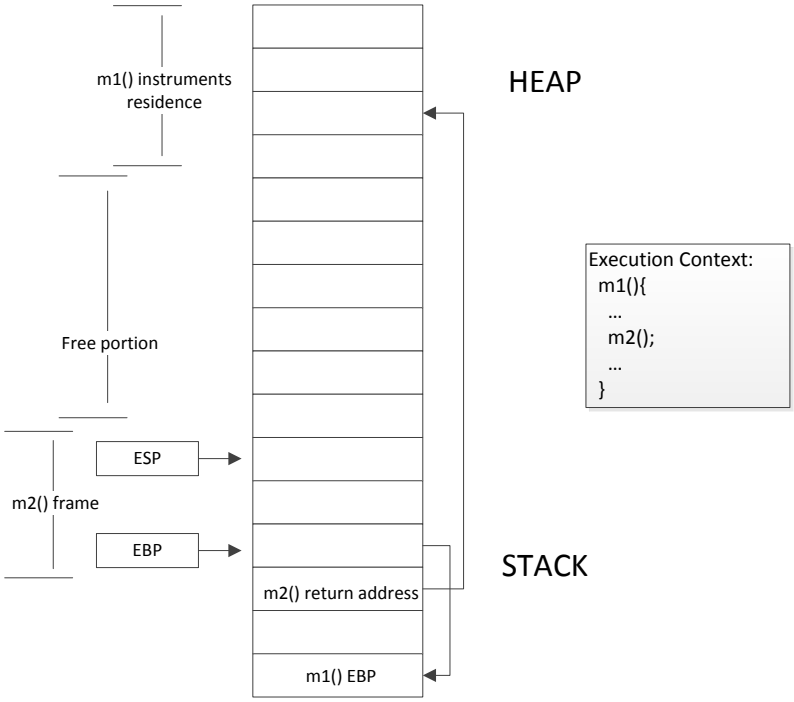


Figure 7.4 Call Stack

Apart from unwinding a call stack by CPU registers, tracking execution context is also an effective approach, however consumes higher overhead. This is done by hooking up two callbacks on before and after occurrence of calls. In Algorithm 2, before every call occur, the hooked function push the being occurred name of the call to the stack, and then pop it when finish. As the call stack is maintained, `thd_stack_` can be accessed in needed to provide the view of execution context.

```
void function UnwindCallStack(address_t ebp, address_t thd_bs,
vector<string> st){
  address_t return_address; RTN rtn;
  while(ebp < thd_bs){
    return_address = (address_t) *(address_t*) (ebp + sizeof(address_t));
    rtn = RTN_FindByAddress(return_address);
    if(!RTN_Valid(rtn)) break;
    st.push_back(RTN_Name(rtn));
    ebp = (address_t) *(address_t*) ebp; // Move backward by dereferece
frame pointer
  }
}
```

Algorithm 1: Call Stack Unwinding

```
map<thread_id_t, vector<string>> thd_stack_;

void function BeforeRoutineCall(THREADID tid, ADDRINT ins_addr){
  RTN rtn = RTN_FindByAddress(ins_addr);
  if(!RTN_Valid(rtn)) break;
  thd_stack_[tid].push_back(RTN_Name(rtn));
}

void funcion AfterRoutineCall(THREADID tid, ADDRINT ins_addr){
  thd_stack_[tid].pop_back();
}
```

Algorithm 2: Call Stack Tracking

7.3 Differentiator – program changes identification

Differentiating the old program with new program can provide grate support in converting a precise schedule. This is to identify the program structural changes between the two versions. The methodology such as UMLDiff [17] can be utilized to identify the changes. However, due to the time limitation of this project, this component will be excluded from this project.

Although the absence of differentiator can affect the performance of RegressionMaple, the online technique in manipulator and similarity measurement technique in convertor can still support the project objective.

7.4 Converter – similarity ranking of execution context

Call stack is a representation of execution context, as shown discussed in Session 7.2. Convert aims at interfacing an outdated inter-thread interleaving instruction schedule to the new version. Considering there are two versions of a program P_1 and P_2. P_1 is the original version and P_2 is the successor. P_1 and P_2 produce same result under an input I.

An iRoot exposed in P_1 is given by $iRoot_{target}(P_1)$ and the corresponding iRoot in P_2 is given by $iRoot_{target}(P_2)$. Convert aims at produce an $iRoot_{target}(P_2)$ under the input $iRoot_{target}(P_1)$.

Consider an idiom 1 iroot,

$$iRoot_{target}(P_1) = (iRootEvent_{a1}, iRootEvent_{a2})$$

$$iRoot_{target}(P_2) = (iRootEvent_{b1}, iRootEvent_{b2})$$

and $iRoot_{target}(P_2)$ is obtained by getting the feasible iRoot in P2 which minimized the distance from every $iRootEvent_a$ in $iRoot_{target}(P_1)$.

Converter utilizes edit-distance to calculate the distance between every $iRootEvent_a$ to a candidate $iRootEvent_b$.The call stack distance between $iRootEvent_a$ and $iRootEvent_b$ is given by$ed_{a,b}$. Call stack of $iRootEvent_a$ is given by $Stack_a = \{a_1, a_2, \dots a_m\}$ and call stack of $iRootEvent_b$ is given by $Stack_b = \{b_1, b_2, \dots b_n\}$.

$$ed_{a,0} = i \; for \; 0 \le i \le m,$$

$$ed_{0,j} = j \; for \; 0 \le j \le n,$$

$$ed_{i,j} = min \begin{cases} ed_{i-1,j} + w_{ins}(b_{i-1}) \\ ed_{i,j-1} + w_{del}(a_{j-1}) \\ ed_{i-1,j-1} + w_{sub}(a_{i-1}, b_{i-1}) \end{cases} for \; 1 \le i \le m, 1 \le j \le n$$

7.5 Manipulator – an naive approach to predict iRoot projection

An online decision making is required to determine whether an instruction execution should be postponed under the requirement of the target iRoot. This is challenging because online scheduler need to make decision while only partial information is given. In this section, the process of manipulator will be presented. It utilizes the collected information and examines a fairly accurate iRoot – a projected iRoot.

The algorithm shown in Figure 7.5 will be involved when an instruction accesses a shared memory location to decide whether it should be postponed. The first iteration may not archive a best matching iRoot, however, the best match will be calculated after the first iteration done. As a result, maximum 2 iterations can archive a best match. The idea will be discusses as follow.

The examination of an iRoot mainly depends on implicitly controlling the execution order of the relevant instructions. As previously discussed that the actual instructions involved in iRoot change across software version, identifying the relevant instructions becomes important. Instructions are typically identified by offset value. It is an image individual ordinal number implied where is the instruction location. This is static information with respected to same binary (image), however changeable with respected to different binaries (images).

When an offset value in the new version is charged, the original schedule becomes inconsistent, thus there is a need to reproduce a schedule. Instead of totally restart from zero, reusing the original information can help to adapt the original schedule. This is an idea of iRoot projection.

```
bool ProjectiRootEvent(UINT32 idx, Inst *inst, thread_id_t curr_thd_id,
     address_t esp, address_t ebp)
 {
   // Obtain the corresponding iRootEvent
   iRootEvent *e = curr_iroot_->GetEvent(idx);

   // (Collector): Collect the execution context
   string a_str = Util::GetCallStack(inst, esp, ebp,
                                     thd_ebp_map_[curr_thd_id]);

   const char *a = a_str.c_str(); // Current callstack
   const char *b = e->st().c_str(); // Targeted callstack

   // (Converter): Calculate edit-distance
   unsigned int ed = ED(a, strlen(a), b, strlen(b));

   // (Converter): maintain schedule
   //              by recording the minimized maximum edit distance
   LockEventStatus();
   if(ed < e->md())
     e->SetMD(ed);
   UnlockEventStatus();

   // (Manipulator): Accept only one pinst at run-time
   if(e->pinst() && e->pinst()!=inst){
     return false;
   }

   // (Manipulator): Determine accept or reject inst under current context
   //                by comparing to minimized maximum edit distance
   bool accept = (ed <= e->md() && ed <= md_);

   // (Manipulator): Update accepted pinst
   if(accept){
     e->SetPinst(inst);
     e->SetPmd(ed);
   }
   return accept;
 }
```

Figure 7.5 - Algorithm of naive iRoot Projection

iRoot projection is based on an assumption that execution context remains similar across versions. This similarity is utilized in Converter's instruction matching mechanism and it can be referred as iRootEvent projection. Manipulator then applies the iRootEvent projection to perform iRoot projection.

Given original iRoot events, manipulator projects the corresponding events' instruction which satisfies:

1. the most closest position (offset) with respect to original instruction;
2. same operator; and
3. the overlapped memory access among (PMem) all projected-instruction.

A pInst - projected-instruction - is an instruction. It is the output of the algorithm and is a replacement of original-instruction in the new binary.

A pw - projection window - is an integer which indicates how far counting from the original-instruction's offset can a projected-instruction be allowed. For example, the original-instruction's offset is 340, and the pw is 10; the allowed projected-instructions are from offset 330 to 350.

As shown in Figure 7.6, projection window is involved in instrumenting suspicious instructions. This method reduces the overhead fairly by only hooking the projection candidates. The instructions which exceed the projection window will not be monitored.

```
// Projection Phase
// Project original instruction to all the new instructions which satify
// (1) same opcode (i.e. same behavior)
// (2) within projection-window
// * explicit callstack checking will be done during runtime
//
ADDRINT img_low_addr = IMG_Valid(img) ? IMG_LowAddress(img) : 0;
for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl);
          bbl = BBL_Next(bbl))
{
  for (INS ins = BBL_InsHead(bbl); INS_Valid(ins); ins = INS_Next(ins))
  {
    ADDRINT offset = INS_Address(ins) - img_low_addr;

    Inst *pinst = GetInst(INS_Address(ins));

    // if instruction matches
    // The distance between current instruction and original instruction
    unsigned int d = (inst->offset() - offset);
    d = (d > 0) ? d : offset - inst->offset();
    if (d < pw_ && inst->opcode() == INS_Opcode(ins))
    {
          // Hook the inst to be monitor
    } // end of if (offset == inst->offset())
  } // end of for ins
} // end of for bbl
```

Figure 7.6 - Algorithm of instrumenting projection candidates

7.6 Implementation strategy

RegressionMaple is implemented by modifying the codes of Maple. This approach can save plenty amount of overhead due to the reduction of interfacing codes. Although the flexibility and maintainability is reduced, code refactoring can be applied in the future to extract the features of RegressionMaple from Maple.

8 RESULTS

In this section, it will show the expected results and actual results of the system. The evaluation methods are also covered.

8.1 Expected Results

The project should improve two situations: (1) incoordination of different versions of same program; (2) inability to assess concurrent bug corrections.

It is possible to improve the performance of concurrent testing by coordinating different versions of same program. In general, a new version of same program often is a partially changed version of the original program. Therefore, a level of the previously generated testing information should be able to convert to the newer version. The converted information will be used in the newer version. This cut the time needed for regenerating the testing information. The size of time cut should depend on the changes which affect the thread interleavings. In an incremental development, this approach should show a significant size of time cut comparing with absence of background information support.

This project is also expected to being able for assessing concurrency bug corrections. The bugs detected in original version are able to provide a narrower set of suspicious interleavings in the newer version. By examining these interleavings, if there is no bug detected, there should be a level of certainty that the bug was fixed by the corrective actions.

8.2 Actual Results

RegressionMaple aims at reproducing a changed iRoot in new version. The following shows a success bug reproduction by projecting an iRoot from its old version.

A general case is shown in Figure 8.1. In the test case, there are two versions of a program; one has concurrency bugs, another try to fix. The test case contributes two testing objectives: (1) can the system adapt an old schedule to the new version; (2) can the system tell if a bug is fixed?

```
40 void *thread(void * num) {          |   40 void *thread(void * num) {
41   unsigned temp = global_count;    |   41   unsigned temp = global_count;
42   temp++;                          |   42   global_count = temp + 1;
43   global_count = temp;             |      --------------------------------
44   return NULL;                     |   43   return NULL;
45 }                                   |   44 }
          main.old.cc                                  main.new.cc
```

Figure 8.1- A general case

Version 1, which is `main.old.cc`, is a buggy old version program and version 2, which is `main.new.cc`, is a new version of the program which is trying to fix the bug. The version 1 has bugs at lines 41 to 43 which require a lock to protect. The developer tried to correct it, however, he made a wrong assumption and the bug still exists.

```
$ org-maple active --mode=run_out --- ./main 2
main: main.cc:36: int main(int, char**): Assertion `global_count==NUM_THREADS' failed.
[MAPLE] === active iteration 1 done === (0.503877)
[MAPLE] active fatal error detected
```

Figure 8.2 - Assertion in main.old

Figure 8.2 shows the assertion error triggered by original Maple.

```
$ org-maple display test_history
24    IDIOM_1 Success 1396822797
```

Figure 8.3 - A bug-triggering iRoot

Figure 8.3 shows the iRoot which is able to trigger the concurrency bug.

```
$ regression-maple active --target_iroot=24 --random_seed=1396822797 --- ./main 2
[MAPLE] === active iteration 1 done === (0.403367)
[MAPLE] active threshold reached
$ regression-maple active --target_iroot=24 --random_seed=1396822797 --- ./main 2
main: main-fix-fail.cc:36: int main(int, char**): Assertion `global_count==NUM_THREADS'
failed.
[MAPLE] === active iteration 1 done === (0.503088)
[MAPLE] active fatal error detected
$ regression-maple display test_history
24    IDIOM_1 Success 1396822797 [                              ] {         }
24    IDIOM_1 Fail    1396822797 [1842, 1855                    ] {1, 2     }
24    IDIOM_1 Success 1396822797 [1846, 1861                    ] {0, 1     }
```

Figure 8.4 - Reproduce the iRoot in new version

Figure 8.4 shows that the iRoot successfully be reproduce in new version in 2nd iteration of active test run.

9 EVALUATION

The evaluation will be done by comparing the performance of RegressionMaple with Maple. The evaluation will mainly focus on the coverage.

First, it will compare the presence and absence of RegressionMaple. In the new bug-fixed version, RegressionMaple will try to execute those iRoots successfully exposed in old version. Maple will try to execute a full run which consists of profile phase and active schedule phase. By comparing this two data, the effectiveness of RegressionMaple can be measured. The measurement will be the coverage identified by Maple deducts the number of success projected-iroots done by RegressionMaple.

Second, it will examine the bug-fixing validation effectiveness of Maple. In the new bug-fixed version, RegressionMaple will try to execute a iRoot which is successfully exposed and able to trigger a concurrency bug in the old version. After two iteration of RegressionMaple, the iRoot corresponding distance should be fairly large. This will reflect that the schedule seems unable to be executed, as this is the effect of locking shared variables. Although this decision should be made by the developer, this value serves as a guidance of confident to bug-fixing effectiveness.

Third, it will examine the invalid-bug-fixing identification effectiveness of Maple. In the new invalid-bug-fixed version, RegressionMaple will try to execute a iRoot which is successfully exposed and able to trigger a concurrency bug in the old version. After two iteration of RegressionMaple, the iRoot corresponding distance should be very close. And the bug should able be triggered. This will reflect that the schedule seems no change, as the iRoot projection should be success. Although this decision should be made by the developer, this value serves as a guidance of confident to bug-fixing effectiveness.

Unfortunately, the setup of benchmarks was fail. Those benchmarks, i.e. Apache, MySQL, a-get, SPLASH2, cannot be compile in environment. This may be due to the incompatibility between the installed packages in the environment and the required packages of those benchmarks. The incompatibility is difficult to resolve and this may be because those versions of benchmarks are legacy. So, the evaluation this report provided only includes the extracted benchmark which is shown in section 8.2.

10 FUTURE IMPROVEMENTS

RegressionMaple inherits some weaknesses from Maple. Maple heavily memorizes the runtime instructions information. Every common memory location access will be recorded; however, this can be reduced. The idea is to cut the unimportant access summaries. As a recent research [18] proved that, the access summaries can be reduced. Applying this idea to RegressionMaple may able to reduce a portion of unfeasible iRoots predicted.

The fail to expose iRoots are unreliable to bring across version. Although some of them can trigger concurrency bugs, projecting the fail to expose iRoots in new version always do not work. This may because these iRoots overlapped with a feasible iRoot. Future study the reason why some fail to expose iRoots can trigger concurrency bugs can utilize this information and provide more precise iRoot projection.

Differentiating the old program with new program can provide grate support in converting a precise schedule. This is to identify the program structural changes between the two versions. The methodology such as UMLDiff [17] can be utilized to identify the changes. However, due to the time limitation of this project, this component will be excluded from this project and remains for further improvement.

11 DELIVERABLES

This project will provide eight deliverables. They are project plan, interim report 1 and release 1, interim report 2 and release 2, final report and final release, and monthly log.

Project Plan: it will show the background information of the project topic, the current situations that this project try to improve, the project objectives, an abstract design to the approach of achieving the project objectives, and the preliminary project schedules.

Interim report 1 and **release 1**: These two deliverables are consistent. Release 1 is a partial implementation of RegressionMaple which is the tool this project proposed. Interim report 1 is reporting the comparison of RegressionMaple and Maple to notify the progressive of the project and the effectiveness of the approach. Although this is a partial implementation, it will provide full essential functionalities. To further explain, release 1 will include all technical components (i.e. stated in section 0), but only basic functionalities. *At least, all skeletons of those components will present and cooperate with each other*: (1) `collector` can extract information in original version which may be inaccurate from `analyzer`; (2) `converter` can accept the information and then write to the testing database of newer version. The information may not facilities to `profiler` execution in this stage; (3) `differentiator` can generate a differentiation report which may be inaccurate to `converter`; (4) `manipulator` can manipulate the `profiler` based on information provided by `converter`, but the information and `manipulation` can be inaccurately.

Interim report 2 and **release 2**: These two deliverables are consistent. Release 2 is an incremental implementation of release 1 to RegressionMaple which is the tool this project proposed. Interim report 2 is reporting the comparison of RegressionMaple and Maple to notify the progressive of the project and the effectiveness of the approach. In this incremental implementation, it will provide more accurate abilities to each function. To further explain, release 2 will refine all technical components (i.e. stated in section 0), but there can still exist inaccuracies on the functions provided. *At least, all technical components will provide more accurate results and show the expected result of the project partially*: (1) `collector` can

extract more accurate information in original version which can beneficial to the new version from `analyzer`; (2) `converter` can partially convert the information extracted from original version to facilitate the new version. The presence of this information should facilities the `profiler` execution in this stage; (3) `differentiator` can generate a differentiation report which is more accurate than release 1 to `converter`; (4) `manipulator` can accept the bug information in original version to manipulate the `profiler` based on information provided by `converter` in the new version. The manipulation should provide certain level of accuracy to the suspicious interleavings.

Final Report and **Final Release**: These two deliverables are consistent. Final Release is an incremental implementation of release 2 to RegressionMaple which is the tool this project proposed. Final report is reporting the comparison of RegressionMaple and Maple to assess the effectiveness of RegressionMaple. In this incremental implementation, it will provide accurate functions. To further explain, final release will refine all technical components (i.e. stated in section 0) to complete the project objectives which are providing regression coverage to concurrent testing and validation on concurrency bug-fixing. *In this stage, all technical components should provide accurate results and show the expected results of the project*: (1) `collector` can extract accurate information in original version which can beneficial to the new version from `analyzer`; (2) `converter` can convert the information extracted from original version to facilitate the new version. The presence of this information should cut the time of `profiler` execution when comparing absence of this information; (3) `differentiator` can generate a differentiation report which is accurate to `converter`; (4) `manipulator` can accept the bug information in original version to manipulate the `profiler` based on information provided by `converter` in the new version. The manipulation should able to examine suspicious interleavings in newer version corresponding to the given bug information in original version. In this final stage, the manipulation should able to show a level of certainty that the bug is fixed when no more suspicious interleavings are detected.

Monthly Log: in each end of month, a brief summary will be provided to keep track of the project progress. The summary will provide the information of critical tasks done and the critical changes of the project.

12 CONCLUSION

This project introduced a new regression coverage driven approach to coordinate two versions inter-thread interleaving instructions schedule. The implementation of regression coverage – RegressionMaple successfully demonstrates the ability of reproduce an interleaving schedule which is changed due to software evolution. The success to expose iRoots can be projected in the new version and reexamine. This reduces plenty of time from retesting the whole system from sketch.

13 REFLECTION

This project is challenging. By the end of it, I have learnt and applied lots of stuff which are either unfamiliar or never used. First, Maple is a Linux based testing tool. It built upon a complex instrumentation library – Intel PIN and record data in high-performance storage – Google ProtoBuf. Maple is programed in C/C++ with Marco use. The user interface is implemented in python and it coordinates the binary file of Maple. The installation includes use of Make to compile the source codes. Maple is a x86-64 instruction-architecture program.

It requires an Ubuntu 12.04 AMD64 operating system; however, execute i386 architecture program. This required alternative environment setup and thus, involved lots of trouble shooting process.

Second, concurrent testing is novel. There is almost no book which discusses or teaches in details. The major resources are research papers. Without strong technical background on foundation of instructions execution, compiler mechanism, and multi-thread mechanism, it is difficult to understand those research papers. In this situation, I studied some materials of that foundation knowledge to build up my understanding on the research papers. This is frustrated. Although there are plenty of related research papers, I did not understand the whole idea of them even after completely read several times. And I was studying the foundation knowledge while did not ensure whether it is really related to the project.

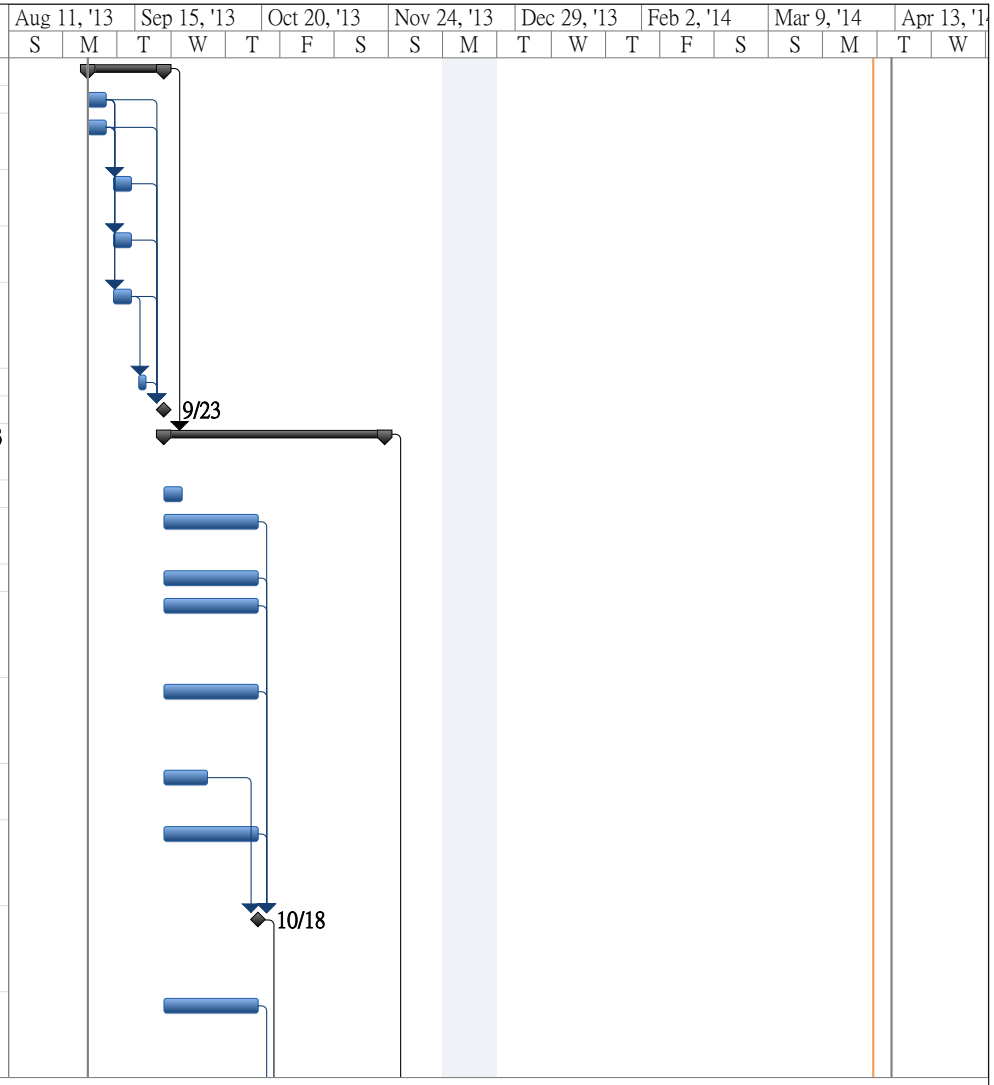
Third, evaluating RegressionMaple requires benchmarks. Open source programs are good to be applied in; however, they are not easy to compile. Compiling them may require addition packages. Some of them may be conflict. Resoling Linux dependencies hell is really painful. I have even removed and installed several version of GNU GCC and compile the GNU GCC from source. This aims at compile a legacy version of MySQL to evaluate RegressionMaple. Although, I have spent many efforts in the evaluation, a lot of candidates still cannot be installed correctly. They are specific version of Apache, MySQL, a-get, SPLASH2, transmission, sqlite, and httrack. Some of them were installed successfully; nonetheless, unable to be involved by Maple and RegressionMaple. Most of the problems are the incompatibility between the environments set up.

Finally, this project although is difficult, I am happy that I could involve in. Beside the technical or academic knowledge, I learnt the learning method which is fairly effective to me. I also being familiar with Linux environment and Linux based development. This is really an attractive platform. This is memorable and meaningful in facing the challenges.

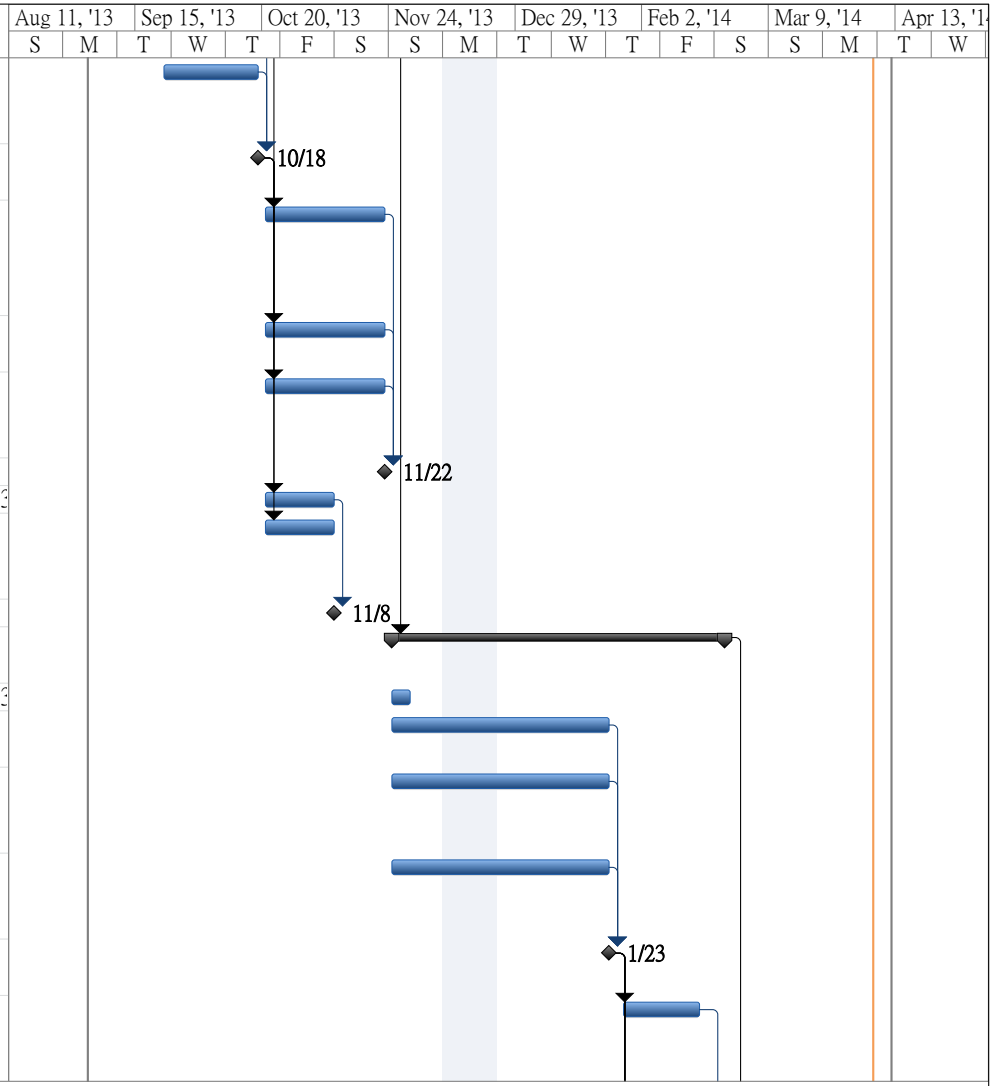
14 PROJECT SCHEDULE

The project will be carried out by an incremental development approach. Each release is an incremental implementation of previous release. The details dependences and the deliverable details can refer to the section of deliverables.

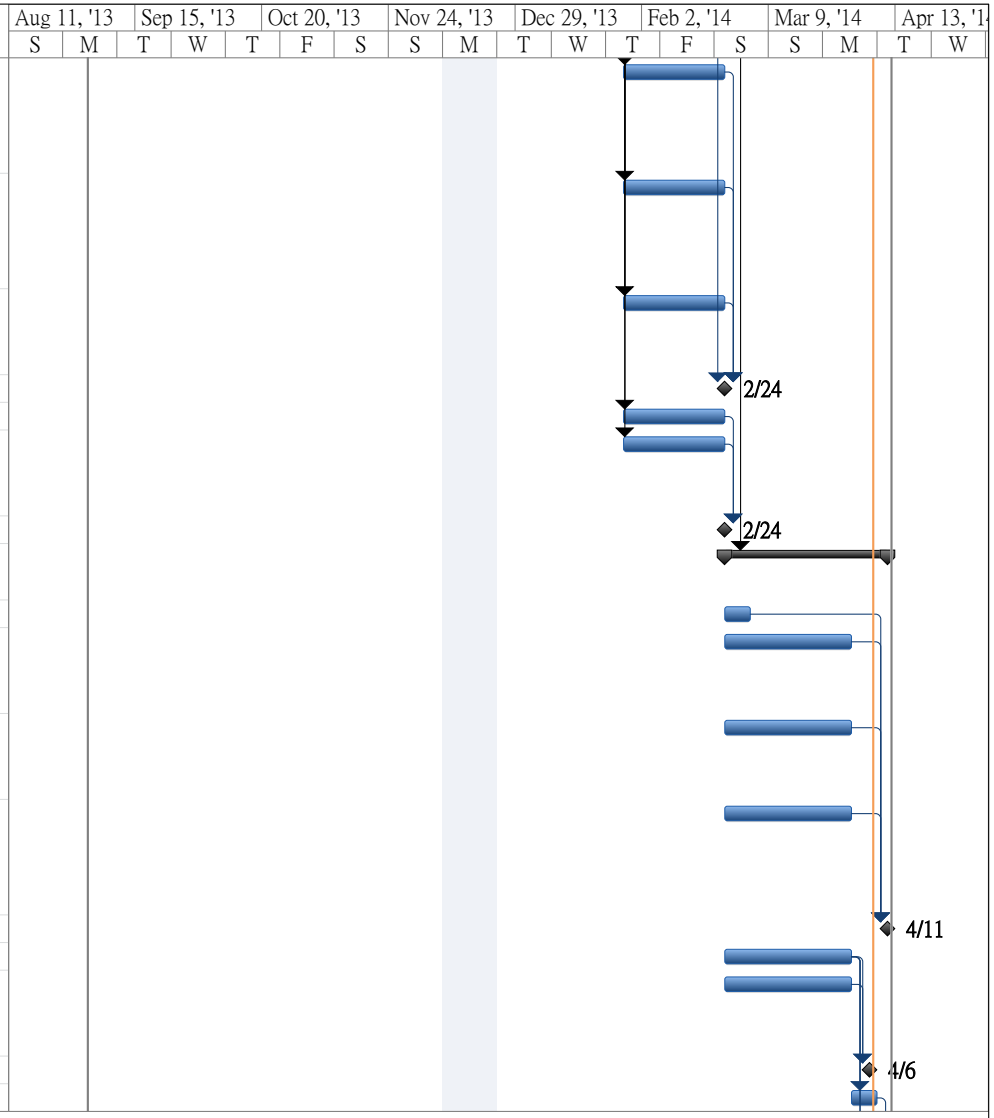
ID	Task Mod	Task Name	Duration	Start
1		**Project Plan**	**15 days**	**Mon 9/2/13**
2		Studying the related literatures	1 wk	Mon 9/2/13
3		Establishing the project objectives	1 wk	Mon 9/2/13
4		Studying the detail nature of the problem	1 wk	Mon 9/9/13
5		Drafting the preliminary solution to the problem	1 wk	Mon 9/9/13
6		Decomposing preliminary solution into technical components	1 wk	Mon 9/9/13
7		Scheduling major tasks	2 days	Mon 9/16/13
8		Project Plan Completed	0 days	Mon 9/23/13
9		**Interim Report 1 and Release 1**	**45 days**	**Mon 9/23/13**
10		Refining the project plan	1 wk	Mon 9/23/13
11		Futher studying the releated literatures	4 wks	Mon 9/23/13
12		Consolidating project objectives	4 wks	Mon 9/23/13
13		Consolidating the understanding of the detail nature of the problem	4 wks	Mon 9/23/13
14		Consolidating the solutionpreliminary solution to the problem	4 wks	Mon 9/23/13
15		Studying the chanllenges of technical components	2 wks	Mon 9/23/13
16		Drafting preliminary algorithms to the technical components' challenges	4 wks	Mon 9/23/13
17		Consolidated domain knowledge and project solutions to problems	0 days	Fri 10/18/13
18		Analyzing component skeletons coordination and basic functionalities	4 wks	Mon 9/23/13



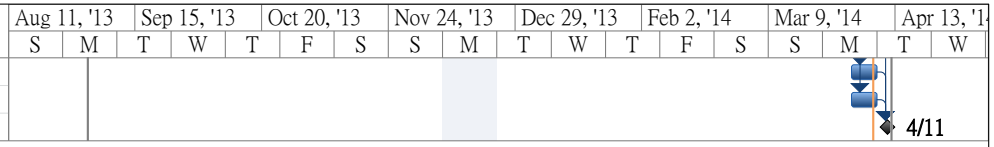
ID	Task Mod	Task Name	Duration	Start
19		Designing, implementing, and testing the skeletons and basic functionalities	4 wks	Mon 9/23/13
20		Component skeletons are implemented	0 days	Fri 10/18/13
21		Further refine the design, approach and implementation of skeletons and the basic funcitonalities	5 wks	Mon 10/21/13
22		Implementing further the basic functionalities	5 wks	Mon 10/21/13
23		Integrating testing and system testing on the skeletons and functionalities	5 wks	Mon 10/21/13
24		Release 1 Completed	0 days	Fri 11/22/13
25		Producing Interim Report 1	3 wks	Mon 10/21/13
26		Prodcuing report of RegressionMaple assesement by comaring RegressionMaple with	3 wks	Mon 10/21/13
27		Interim Report 1 Completed	0 days	Fri 11/8/13
28		**Interim Report 2 and Release 2**	**30 days**	**Mon 11/25/13**
29		Refining project plan	1 wk	Mon 11/25/13
30		Analyzing the approach to provides further accurate results	3 wks	Mon 11/25/13
31		Refine the design of technicial components to support further accurate results	3 wks	Mon 11/25/13
32		Implementating and testing the enhancement of providing further accurate results	3 wks	Mon 11/25/13
33		Further acurrate results are provided	0 days	Thu 1/23/14
34		Analying the apporach to providing expected result of the project partially	2 wks	Tue 1/28/14



ID	Task Mod	Task Name	Duration	Start
35		Refineing the desing of technicial components to provide expected result of the project partially	3 wks	Tue 1/28/14
36		Implementing and testing the modification of the technical components to provide expecte result of the proejct partially	3 wks	Tue 1/28/14
37		Further integration testing and system testing on the newly implmented codes	3 wks	Tue 1/28/14
38		Release 2 Completed	0 days	Mon 2/24/14
39		Producing Interim Report 2	3 wks	Tue 1/28/14
40		Prodcuing report of RegressionMaple assesement by comaring with Maple	3 wks	Tue 1/28/14
41		Interim Report 2 Completed	0 days	Mon 2/24/14
42		**Final Report and Final Release**	**33 days**	**Tue 2/25/14**
43		Refine project plan	1 wk	Tue 2/25/14
44		Analying the approach to provide accurate results and expected result	5 wks	Tue 2/25/14
45		Refine the design of technicial components to support accurate results and expected result	5 wks	Tue 2/25/14
46		Implementating and testing the enhancement of providing accurate results and expected result	5 wks	Tue 2/25/14
47		Final release Completed	0 days	Fri 4/11/14
48		Producing Final Report	5 wks	Tue 2/25/14
49		Prodcuing report of RegressionMaple assesement by comparing with Maple	5 wks	Tue 2/25/14
50		Final Report Completed	0 days	Sun 4/6/14
51		Preparing presentation sides	1 wk	Tue 4/1/14



ID	Task Mode	Task Name	Duration	Start
52		Preparing presentation scripts	1 wk	Tue 4/1/14
53		Practicing presentation	1 wk	Tue 4/1/14
54		Project Presentation Completed	0 days	Fri 4/11/14



15 REFERENCES

[1] M. Pezzè and M. Young, *Software testing and analysis : process, principles, and techniques*. Hoboken, NJ: Wiley, 2008.

[2] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: A Coverage-Driven Testing Tool for Multithreaded Programs," in *OOPSLA*, 2012.

[3] S. Kadry, "On the improvement of cost-effectiveness: A case of regression testing," in *Advanced automated software testing : frameworks for refined practice*, I. Alsmadi, Ed. Hershey PA: Information Science Reference, 2012, pp. 68-88.

[4] S. Park, S. Lu, and Y. Zhou, "CTrigger: Exposing Atomicity Violation Bugs," in *ASPLOS*, Washington, DC, 2009.

[5] P. Godefroid, "Model Checking for Programming Languages using VeriSoft," in *POPL*, 1997, pp. 174-186.

[6] M. Musuvathi and S. Qadeer, "Iterative Context Bounding for Systematic Testing of Multithreaded Programs," in *PLDI*, California, 2007.

[7] M. Musuvathi, et al., "Finding and reproducing heisenbugs in concurrent programs," in *PLDI*, 2008.

[8] K. Poulsen. (2004, Feb.) Software Bug Contributed to Blackout. [Online]. http://www.securityfocus.com/news/8016

[9] K. Sen, "Race directed random testing of concurrent programs," in *PLDI*, 2008.

[10] W. Zhang, C. Sun, and S. Lu, "Conmem: detecting severe concurrency bugs through an effect-oriented approach," in *ASPLOS*, 2010.

[11] F. Sorrentino, A. Farzan, and P. Madhusudan, "PENELOPE: Weaving Threads to Expose Atomicity Violations," in *FSE*, 2010.

[12] B. Lucia, J. Devietti, K. Strauss, and L. Ceze, "Atom-Aid: Detecting and Surviving Atomicity Violations," in *ISCA*, 2008.

[13] D. Engler and K. Ashcraft, "RacerX: Effective, Static Detection of Race Conditions and Deadlocks," in *SOSP*, 2003.

[14] D. Deng, W. Zhang, and S. Lu, "Efficient Concurrency-Bug Detection Across Inputs," in *OOPSLA*, Indianapolis, Indiana, 2013.

[15] H. K. N. Leung and L. White, "Insights into Regression Testing," in *Proc. Conf. Software Maintenance*, 1989.

[16] C.-K. Juk, et al., "Pin: Building Customized Program Analysis Tools with Dynamic

Instrumentation," in *PLDI*, 2005.

[17] Z. Xing and E. Strouliz, "UMLDiff: an algorithm fo object-oriented design differencing," in *20th IEEE/ACM international Conference on Automated software engineering*, New Youk, 2005.

[18] C. Yan and C. W.K., "Lock Trace Reduction for Multithreaded Programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, pp. 2407-2417, 2013.

[19] G. J. Myers, *The Art of Software Testing*, 2nd, Ed. Hoboken, New Jersey: Wiley, 2004.

[20] T. Apiwattanapong, A. Orso, and M. J. Harrold, "JDiff: A differencing technique and tool for object-oriented programs," in *ASE*, Linz, Austria, 2004, pp. 3-36.

[21] V. Jagannath, M. Gligoric, D. Jin, G. Rosu, and D. Marinov, "IMUnit: Improved Multithreaded Unit Testing," in *IWMSE*, Cape Town, South Africa, 2010.

[22] B. Li, Y. Wang, and L. Yang, "Programs, An Integrated Regression Testing Framework to Multi-Threaded Java," in *Software Engineering Technique: Design for Quality*, K. Sacha, Ed. Boston: Springer, 2006, pp. 237-248.

16 MONTHLY LOG

Month	Log
March 2014	Apply Maple C++ test program, which is sharedCounter.cc, to evaluate the system whether can run an old execution schedule in a new version of the same program Apply SQLite to evaluate the differences of Maple and RegressionMaple
Feb 2014	Implement both predictor and observer by using callstack unwinding approach to retrieve execution context Instruction window is applied to deal with changes of program, that is, code changes during software evolution. Store callstack information of each exposed iroot in Google Protobuf format; Use protobuf to transfer information between predictor and observer as well as across software versions.
Jan 2014	Completed Interim Report 2. Studied some related aspects and performed some specific tasks: Found that Maple used IMG's path (i.e. binary file's path) for identifying iRoot event. But the name may change across software evolution, for example, changed from //pro/v1/main to //pro/v2/main. Studying DWARF and Libdwarf.h to collect information (execution context) of a running testee by given the address of instruction. Designed an approach to convert schedule from original version of a testee to a new version. Instead of comparing every part of instruction address, comparing the execution context is suitable to convert schedule. To record a schedule, RegressionMaple will drive as Maple by Intel PIN, retrieve Debugging Information for collecting information of execution context, and then store event (e.g. read or write a shared variable) by Google Protobuf. To convert a schedule, RegressionMaple will read every stored event, which stores in Google Protobuf format, and on the other side, drive as Maple by Intel PIN, retrieve Debugging information to build execution context, and then enable or disable a thread according to the comparison result of execution context.

Dec 2013	Studied the detailed flows of MAPLE to decide an approach to extend it Studied Intel PIN instrumentation API to decide which approach of program's flows is considerable Designing the program's detailed flows Implementing the program
Nov 2013	started implementation of the project studied feasibility of various approaches to implement the project further revised the algorithm of regression coverage
Oct 2013	Completed Interim Report 1 Refined problem definition and scope Studied major alternatives, current status and limitation Refined the project objectives Refined the project schedule Established a high level design of the project to solve the stated problems