
Concurrency in a Decentralized Automatic Regression Test Selection Framework for Web Services

Michael Ruth, Ph.D.

Lock Haven University of Pennsylvania

Department of BA, CS, & IT

mruth@lhup.edu



**LOCK
HAVEN**
UNIVERSITY
OF PENNSYLVANIA

Overview

- Web Services
- Regression Testing
 - Regression Test Selection
 - Safe RTS Techniques
- Main Contributions
- Safe RTS technique for Web Services
 - Overview
- RTS and Concurrency Concerns
 - Supporting Scenarios
 - Coverage Conflict
 - Meeting Concurrency Concerns
- RTS Algorithms
- Conclusion
- Future Work

Web Services

- Are quickly becoming the de-facto standard for modeling *business processes*
- Are based on internet standards such as HTTP, SOAP, WSDL, UDDI, etc.
 - Enjoy widespread use largely to both their
 - *Interoperability*
 - *Composability*
 - Additionally, they are by nature *distributed* and can be thought of as *developed autonomously*

Regression Testing

- ***Regression Testing*** (RT) is the execution of a battery (group) of test cases on a program in order to ensure that the modification most recently made does not produce unintended faults
 - *Test case* - program input and expected output
 - *Test suite* - set of test cases
 - *Coverage metrics* (branches or statements the test cases cover)

Regression Testing Process

- Five basic problems associated with RT
 - Test case revalidation problem
 - Identify and remove obsolete test cases from T when specifications change
 - ***Regression test selection problem***
 - ***Select $T' \subseteq T$, a set of test cases to execute on P'***
 - Test suite execution problem
 - Test P' with T' , establishing P' 's correctness with respect to T'
 - Coverage identification problem
 - If necessary, create T'' , a set of new test cases for P'
 - Test suite maintenance problem
 - Create T''' , a new test suite and execution profile for P' , from T , T' , and T''

Safe RTS

- Safe techniques offer an important new criteria
 - The selected subset of T' contains all test cases in T that can reveal faults in P'
 - This implies that safe techniques *provide the same level of confidence as removing no test cases* at all
- Safe Regression Testing is generally *code-based* testing
 - CFG approaches follow three basic steps:
 - Constructs a Control-flow graph representation of code
 - Finds dangerous edges (through comparison)
 - Selects test cases (using coverage matrix, CFG, and dangerous edges)
 - *Safe RTS approaches cannot be directly applied* to Web services

Related Work

- Tsai, et al. Scenario Based model
 - End-to-end RTS for Web Service frameworks
 - *Unsafe (scenario slicing)*
- MJH, et al, Component Based Model
 - Safe RTS for Component based systems
 - *Technique cannot be composed, no notification system, centralized authority*
- Mansour, et al. TLTS-based RTS
 - RTS technique for WS frameworks
 - Composable & safe
 - *Centralized authority, specification-based, and Information Sharing issues*

Main Contributions

- The proposed approach is a novel safe RTS technique for the verification of Web services which is:
 - *end-to-end, automated, distributed*
 - *interoperable* and *composable*
- This work is novel because it is the first which:
 - *Recognizes* and *solves issues* related to *concurrent modifications in a decentralized RTS framework for Web services*

Safe RTS Approach for WS

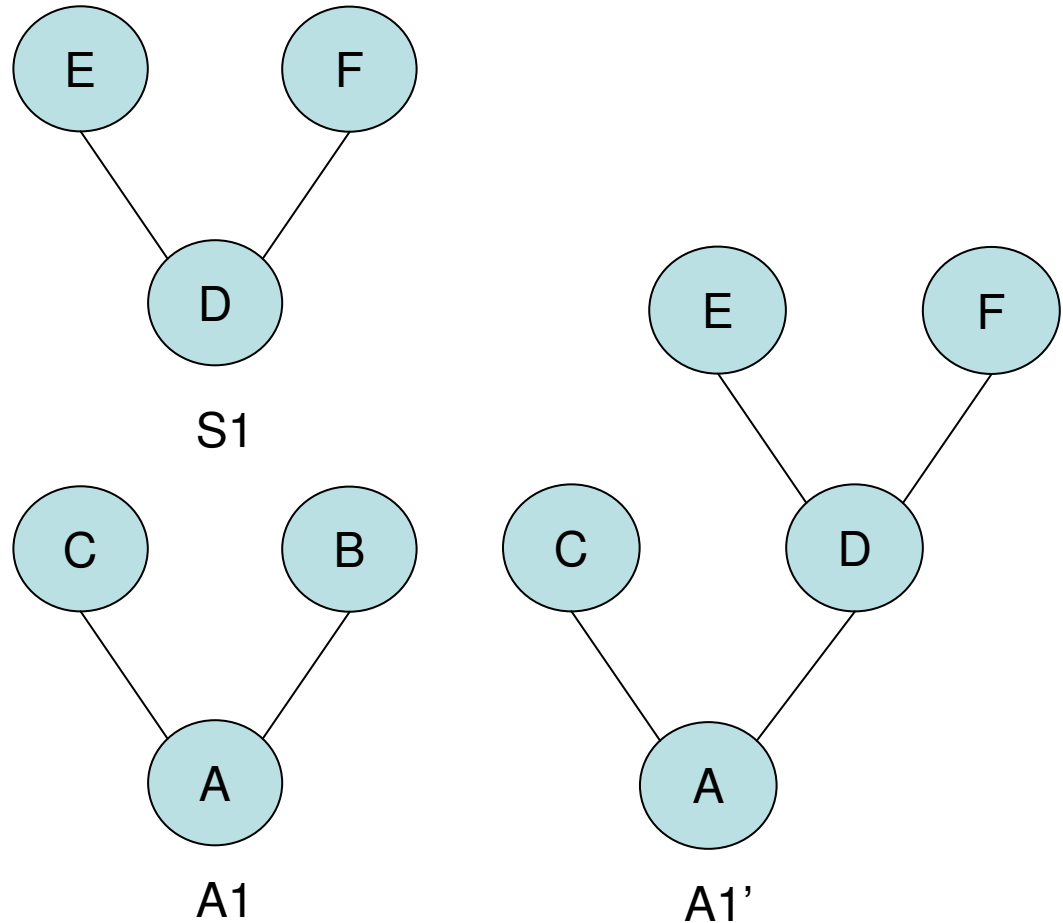
- 3-step approach of CFG-based RTS which has two distinct phases of operation
 - *Initialization Phase* (simple & composite)
 - Initializing three required elements
 - Control-flow graphs
 - Test Cases
 - Coverage Information
 - *Critical Phase*
 - *Finding dangerous edges and selecting tests*
 - Same algorithms as those used in traditional monolithic apps

Construction Phase

- Construct CFG
 - control flow graphs are composed as the services themselves are composed
 - WSDL changes are ignored
 - glue code is ignored (are libraries which do NOT change if WSDL doesn't)
 - Two specific tactics are employed to ensure a higher level of participation:
 - the *code artifacts are shielded* from the testers
 - textual artifacts are encoded using a “one-way” hash
 - The *granularity of the published CFG* is at the *discretion of the service provider*
- Construct test suite and coverage information

Construction Phase: Example

- Suppose that we have an application (A1) and a service (S1) →
- Suppose also that A1's B is where A1 calls S1
- A1 calls S1, gets its CFG and places it where B would be (A1')
- S1 and A1' are terminal (complete) graphs and that A1 is a composed graph



Critical Phase (automation)

- A set of *distributed agents* will interact together
 - One for each point of test responsible for
 - *Notifying interested parties* of system modifications
 - There is a notification list to handle this functionality which is built during construction phase
 - *Performing Regression Test Selection* when notified
 - Builds a CFG on the fly to compare to the stored CFG
 - *Running the tests* once RTS process is completed

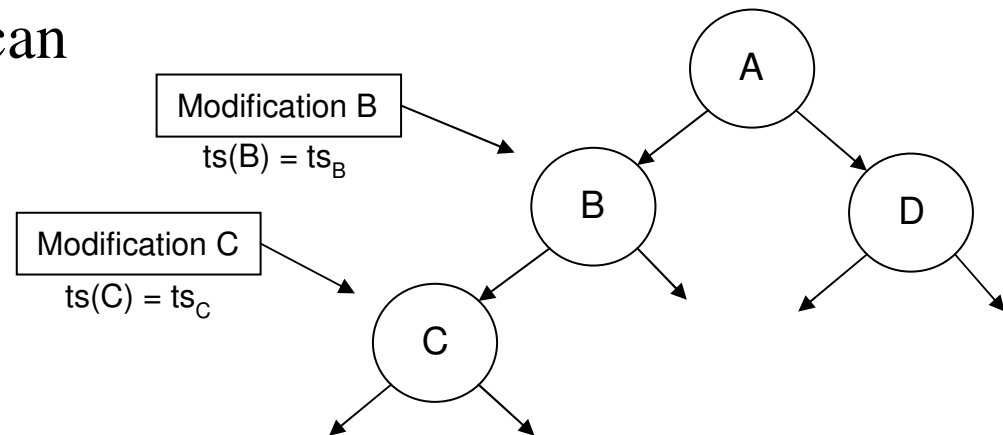
Concurrency Concerns

- Automating the proposed approach presents an entirely new set of issues
 - *concurrent modifications* become increasingly important
- These issues are:
 - *Fault Locatability*
 - *Test Inconsistency*
 - *Communication issues*
- A set of scenarios will help to illuminate these issues

“Call Graphs”

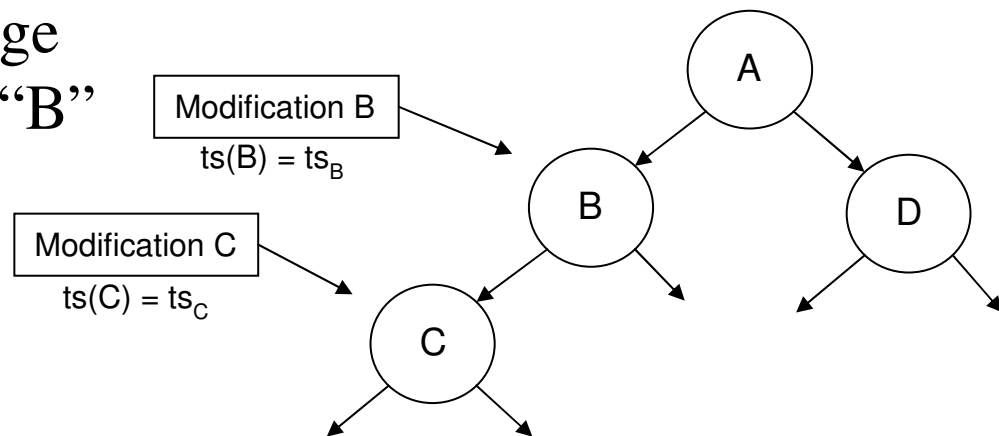
- We need to quickly discuss how to discuss agent interconnection
 - And we do this with call graphs...
 - The nodes in this diagram are services (which each have an agent)
 - the edges form a “can call” relationship

Note that no Agent has a complete “call graph”



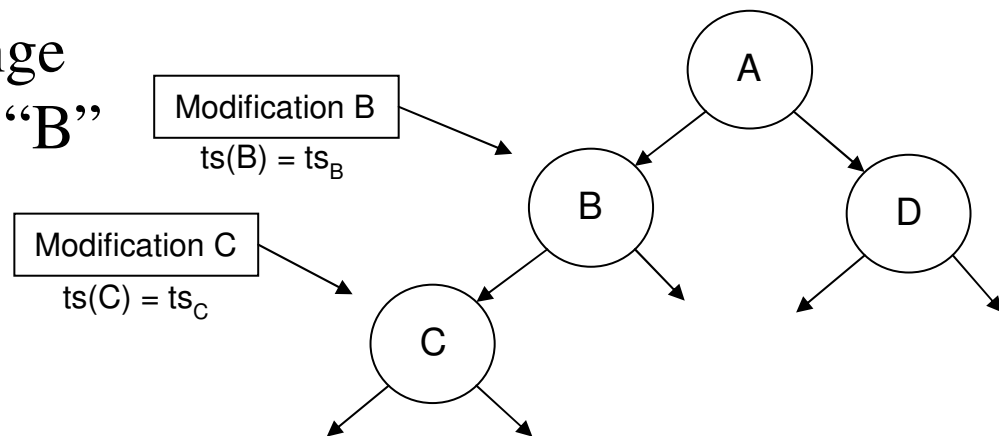
Scenario I

- Considering the figure below:
 - Suppose node A first receives a change notification from “C” with a $ts(c) = 10$
 - A selected the test cases and began testing
 - While A was testing C:
 - A receives a change notification from “B” with $ts(b) = 12$



Scenario II

- Considering the figure below:
 - Suppose node A first receives a change notification from “C” with a $ts(c) = 10$
 - A selected the test cases and began testing
 - While A was testing C:
 - A receives a change notification from “B” with $ts(b) = 5$



Coverage Conflict

- *Coverage Conflict* is an important component of Fault Locatability & Test Inconsistency
 - Since *the technique depends on the locations* of the modifications
 - One modification can *cover* another
 - More formally:
 - *Two modifications, M_1 and M_2 are in coverage conflict if the regression test selection algorithm selects some common test cases.*
 - *In other words, M_1 coverage conflicts with M_2 if $T_1 \cap T_2 \neq \emptyset$.*

Meeting the Concurrency Challenges

- Fault locatability requires that coverage conflict be prevented
 - Too restrictive
 - Requires strict synchronization and total ordering
 - Test inconsistency
 - To ensure test consistency, one must also ensure fault locatability
- ***Stable-State, or Eventual, test consistency***
 - Once the system reaches a stable state, the last set of test cases is guaranteed to be consistent
 - Allows for a greater amount of concurrency without requiring strict synchronization

RTS Algorithms

- Observing the discussed anomalies, *a set of algorithms was developed* to control the distributed RT and RTS processes
 - The agents follow a set of three predefined algorithms *which guarantee eventual test consistency*
 - A tester object is also employed to run test cases
 - Two categories:
 - Support:
 - *TestCase[] selectTestCases(EdgeSet es);*
 - *coverageConflict(EdgeSet a, EdgeSet b);*
 - *mergeDangerousEdgeLists(EdgeSet a, EdgeSet b);*
 - Event Handlers:
 - *localModification()*
 - *receive(MSG)*

Tester & Support Algorithms

- The *Tester* object actually runs the test cases:
 - *Tester.test*
 - Parameters: A set of Test Cases
 - Operation:
 - As it runs test cases it moves the finished test cases from a `to_test` queue to a `done` list
- *Support Algorithms:*
 - *TestCase[] selectTestCases(EdgeSet es);*
 - Selects test cases based on EdgeSet
 - *boolean coverageConflict(EdgeSet a, EdgeSet b);*
 - Determines if two tasks conflict based on coverage
 - *EdgeSet mergeDangerousEdgeLists(EdgeSet a, EdgeSet b);*
 - Merges two dangerous edge lists so that if a new task conflicts with either it will conflict with the result

Agent localModification()

- Upon a local modification:
 - the *global CFG* of the local site *is updated*
 - *A message is created* (including the id, logical clock, and CFG of the operation)
 - This *message* is then *sent to all subscribers*
- Note that *RT and RTS are not performed* here
 - The service providers are responsible for this

Agent receive(MSG)

- Upon *receiving a new modification*:
 - The message is *checked against watermark for ordering*
 - If lower than watermark, discard
 - Update the watermark if higher than last seen
 - Build new CFG by embedding MSG.CFG into old CFG
 - Build and send new message using new global CFG as performed in localModification()
 - *Compute Dangerous Edge List by comparing old and new CFGs*
 - Select test cases using dangerous edge list
 - Create a new *Tester* (task)
 - using dangerous edge list and selected test cases

Agent receive(*MSG*) (*cont*)

- *Determine if the incoming task conflicts with any running tasks*
 - *If they do, the conflicting tasks are merged and the running task is terminated*
 - The merged result is then tested against the remaining tasks
 - *If none of the tasks conflict with the incoming task (or all conflicts have been resolved) the newly generated task is started*
- *Incoming task is added to set of running tasks*

Merging Testers

- In order to successfully merge two testers:
 - Three local variables must be updated:
 - The “to-do” list =
 - $\text{Tester}' . \text{to_test} \cup \text{Tester}_c . \text{to_test}$
 - The “done” list =
 - $\text{Tester}' . \text{done} \cup (\text{Tester}_c . \text{done} - \text{Tester}' . \text{to_test})$
 - $E' = \text{merging of } E_c \text{ \& } E'$

Conclusion

- Our approach solves the how of *Safe RTS to be performed across the enterprise* at any point in the enterprise that :
 - Is *distributed, automated, and end-to-end*
 - Solves the issues related to *interoperability & composability*
- Additionally, the issues related to concurrent modifications were *recognized* and *solved* in the form of *distributed agents* and a *set of algorithms*
 - *Ensuring Eventual Test Consistency*

Future Work

- WSDL modifications are ignored
 - In practice, they cannot be
 - Augment existing system to include WSDL monitoring and notification scheme
- Static Composition
 - The present scheme is limited to static composition
 - Augment the framework with a UDDI component
- Ensuring Fault Locatability & Test Consistency
 - More restrictive algorithms and tighter synchronization
- Data Flow Regression Test Selection Framework
 - May not be feasible due to information requirements

Questions?



Empirical Analysis of Approach

- Performed to *demonstrate* that the proposed *approach is feasible* and *can be cost effective*
 - The analysis will be based on a previous empirical analysis of traditional monolithic applications
 - Performed over a set of systems (benchmark)
 - Systems randomly modified
 - 30 test cases per path
- *Five systems* which *are representative of real-world* Web services *were developed for this purpose*
 - serve as a *benchmark for the evaluation of the effectiveness of RTS techniques* for the verification of WS frameworks

Evaluation of Empirical Analysis

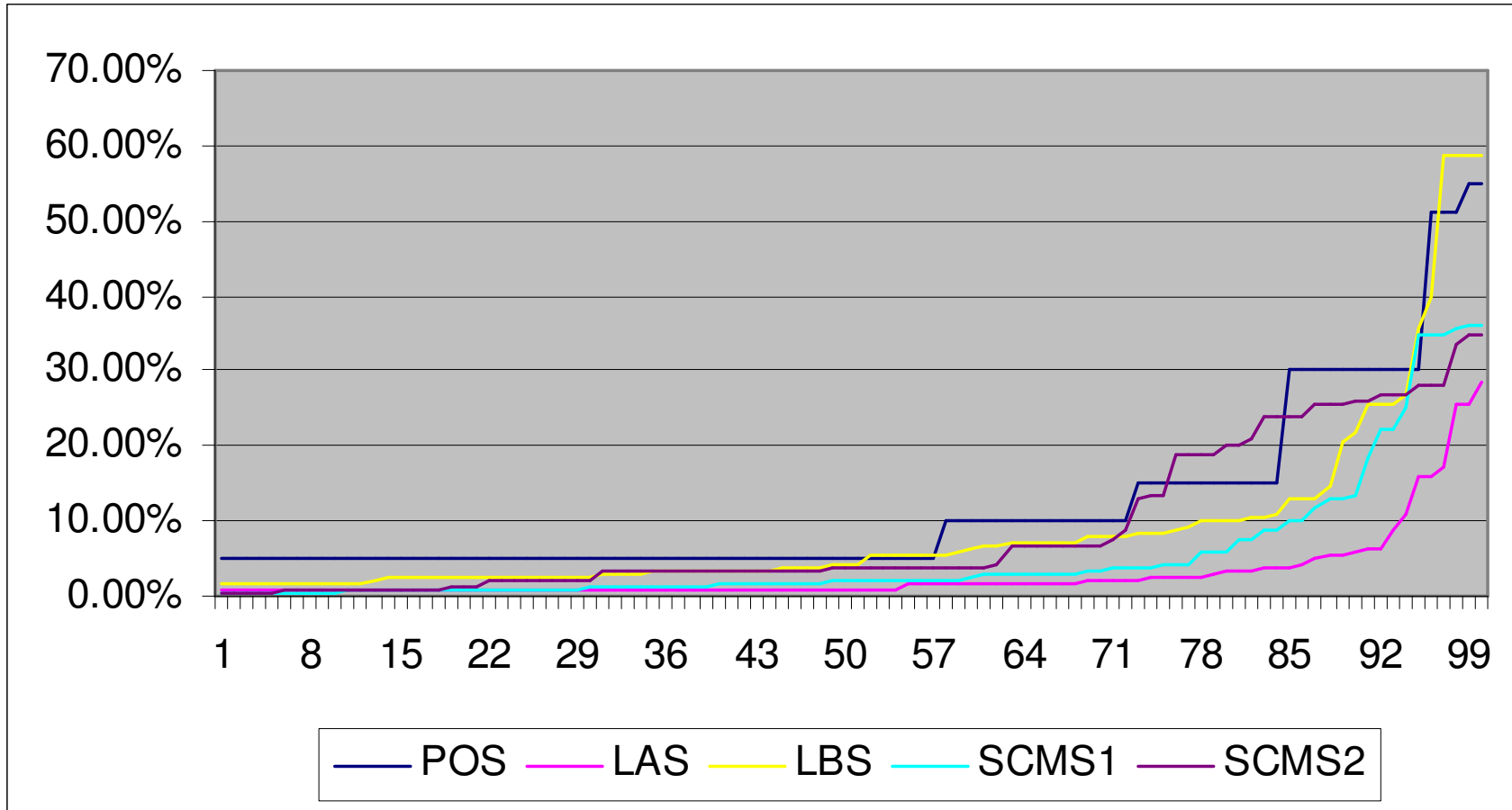
- The *cost of performing all tests* throughout the system
 - will be compared with
- The *cost of performing the regression test selection and the cost of performing only the selected tests* throughout the system

Experimental Setup and Operation

- Each of the systems was first augmented with test cases, coverage information, and control-flow graphs
- Three steps are then followed:
 - the test harness *randomly selects a node* and *modifies it*
 - *regression test selection is performed for each of the affected services and the cost is recorded.*
 - the *harness ran the selected test cases* from the second step and *recorded the time required for the entire set to run*



Results of Empirical Analysis





Discussion and Conclusion of Empirical Analysis Results

- On Average (across systems), the results of the empirical analysis in terms of cost savings are:
 - Worst: **57%**
 - Mean: **88%**
 - Median: **97%**
- From the analysis, ***meaningful cost savings were achieved*** which implies that this ***approach can be cost effective!***