Software Analytics by Storage (SAS)

Sundaresan Nagarajan – McAfee/Intel India

Sundaresan_Nagarajan@mcafee.com

Abstract

There are a number of challenges in software, especially in its effective testing and the software maintenance problems. Can there be something done to improve the testing efficiency and help deal with the software maintenance problems? This paper presents a case that software analytics based on a storage-based approach can help deal with these issues. This database storage-based system along with simple graph-based queries, applies to different types of software, platforms, domains and also to complex multi-tier software.

This system works by making use of a lower level view of the software and mapping it to high level user-domain features and tasks with the help of a database. For example, when a test case is executed, the test case gets tagged to the path traversed by it in the structural graph of the software which is stored in a database. The objective is to exploit the inherent knowledge present in the software to improve its quality. There are two parts in this solution: a minimal database-based storage layer that acts as a base for overlay applications which solve a specific problem. There are two such overlay applications discussed here: parallelized testing and automated root-cause analysis.

The minimal storage layer consists of a relational database which is populated with both static and runtime behavior of the software. This storage layer is queried by overlay applications working on graph based queries to solve a specific problem like: the right set of test cases to execute for a bug/feature, parallel testing (concurrent execution of test cases), root-cause analysis for a bug or test case design

Some of the benefits of this system are that it helps right from software development testing to software maintenance post release. The relational database which is part of the storage layer matures with time based on testing done and issues found in the field deployment which is looped back into the system. As we close the loop with every issue found in the field and testing of the software, the system gets better as new software features and fixes are introduced. Some of the other benefits of this system are identifying parallel test cases so that better understanding and more efficient testing is possible, and also efficient and accurate root cause analysis based on depth querying and graph traversal algorithms. It is a step towards automated root cause analysis.

On the whole, the storage approach captures the dynamic relational knowledge of software and its execution. The silos of information, that is code logic at different layers of the software, are brought together in a relational database to make a combined sense. This offers greater insight into the software, thus increasing its efficacy.

Biography

Sundaresan Nagarajan is a Senior Software Engineer in Test working at McAfee/Intel India. Prior to this, he has worked in the storage domain at EMC Corporation. He has around 6 years of experience working in the software industry in both the fields of development and software testing. He has resolved over a 100 software customer issues reported from the field and also worked on software development projects initiated from scratch. He is a graduate in Computer Science and Engineering from Anna University, Chennai, India.

1. Introduction

Is there something that can be done to improve the software testing process and the numerous software maintenance problems? Are developing code, software testing and software maintenance related? Why don't we avert software maintenance problems and increase software maintenance efficiency with time?

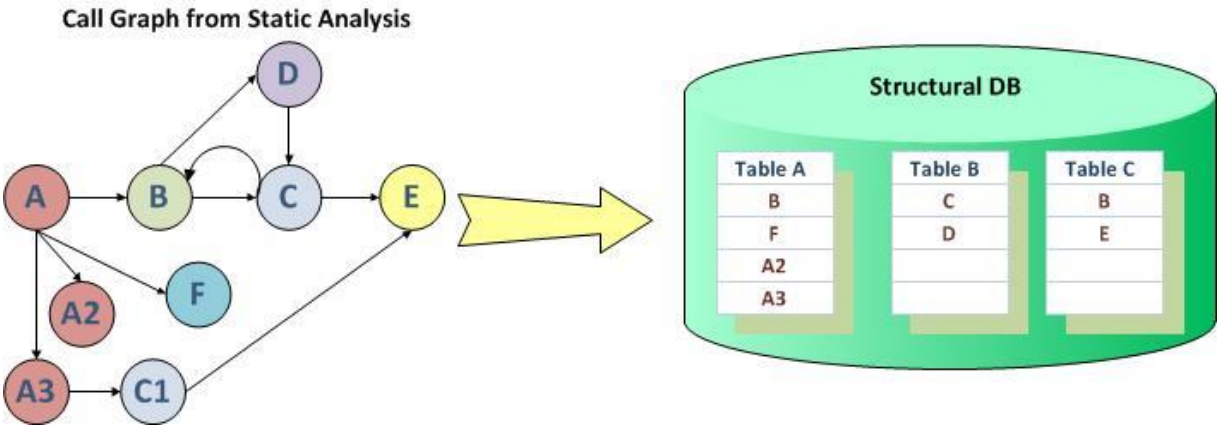
These are the some of the questions, which led to the Software Analytics by Storage (SAS) system. The objective of this system is to exploit the inherent knowledge of the software to improve its quality by improving the testing process and help in solving the numerous software maintenance problems.

This paper proposes a storage and database based solution to improve the software quality. The database system proposed here matures over time. With issues root-caused, testing already done need not be redundant and issues are averted.

From the user perspective, the SAS system provides a simple user query system. Complex applications like graph search can be built on top of it.

2. Database and Storage based approach

The approach to solving the problems mentioned here is take the output data from static analysis software (like Doxygen) and its call graph (Ryder 1979, Grove 1997, Callahan 1990) details and store them in a database. Also the computations performed on the stored static data are captured in a database which enables subsequent user querying. This forms the central theme of the storage-based approach. This approach is illustrated by the diagram below.



The relational database identifies the relationships between the data records of a set of data, commonly referred to as tables. The referential integrity property of databases ensures that the correct relationships are formed as the data is fed into the database.

The other major advantage of a database approach is its ability to mature with time, as relationships that have been captured gets augmented as new features and fixes are added, and new issues and testing related static data are fed into the system. This approach is capable of helping in applications like root cause analysis. Even during the pre-release phase of the software, the maturing aspect of the storage-based approach ensures that more effective identification of complex issues can be done while testing. This will ensure that the customer gets a high quality product at first usage.

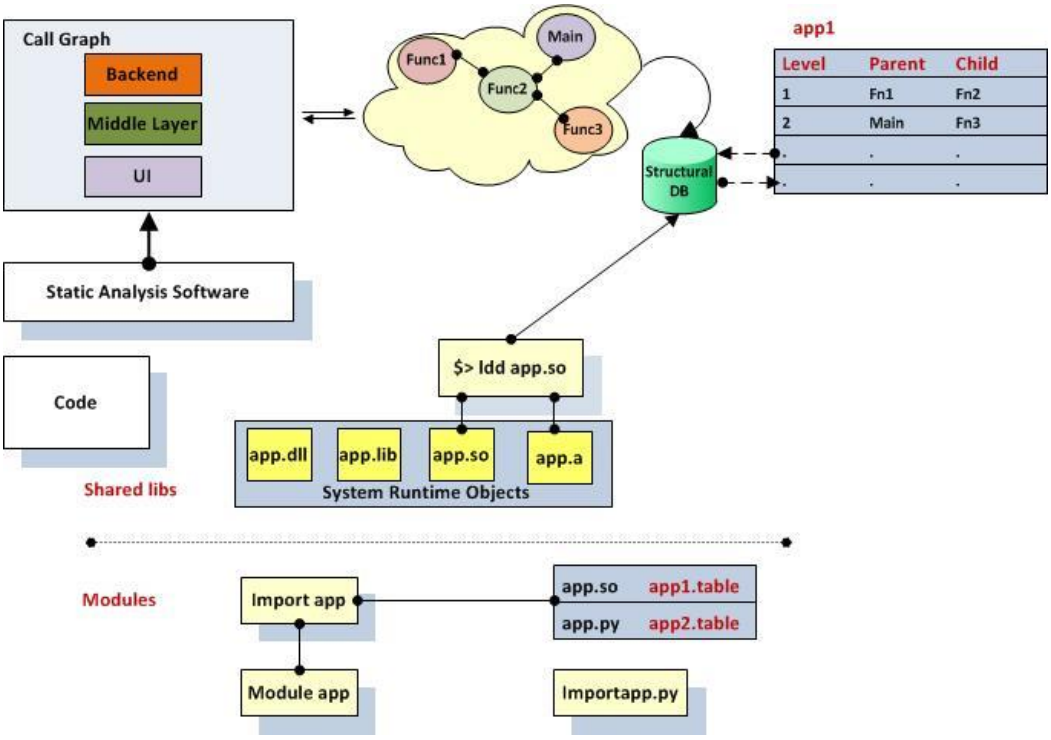
Now that we have an overview of the database and storage based approach to software testing and maintenance, in the next section we can look at how a typical storage base structural database is modeled.

3. Concept of Structural Database

The structural database captures the structural details of the software being analyzed. Subsequently the run time data of the software (like debugger recordings/stack trace analysis (Gavrilov 2000)) of the software features are linked with the structural database to form the relational mapper database. This association between the structural database and the runtime captured information takes place by common key and record attributes.

The figure below illustrates this concept. The software static analysis generates the methods and the call sequences. The structural database links the method calls associated with a library. The information in the structural DB contains not only the call graph information but also libraries and system runtime objects (.dll, .lib, .so, .a) across platforms and across the languages (like .cpp and .py) which are the components of a multi-tiered software system.

This is useful for the system as we go through an end-to-end scenario of testing, runtime analysis, etc where the structural DB contents such as the call graph of code, libs, and modules could be part of a single test-case path.



4. Change set and Sequence of Trees

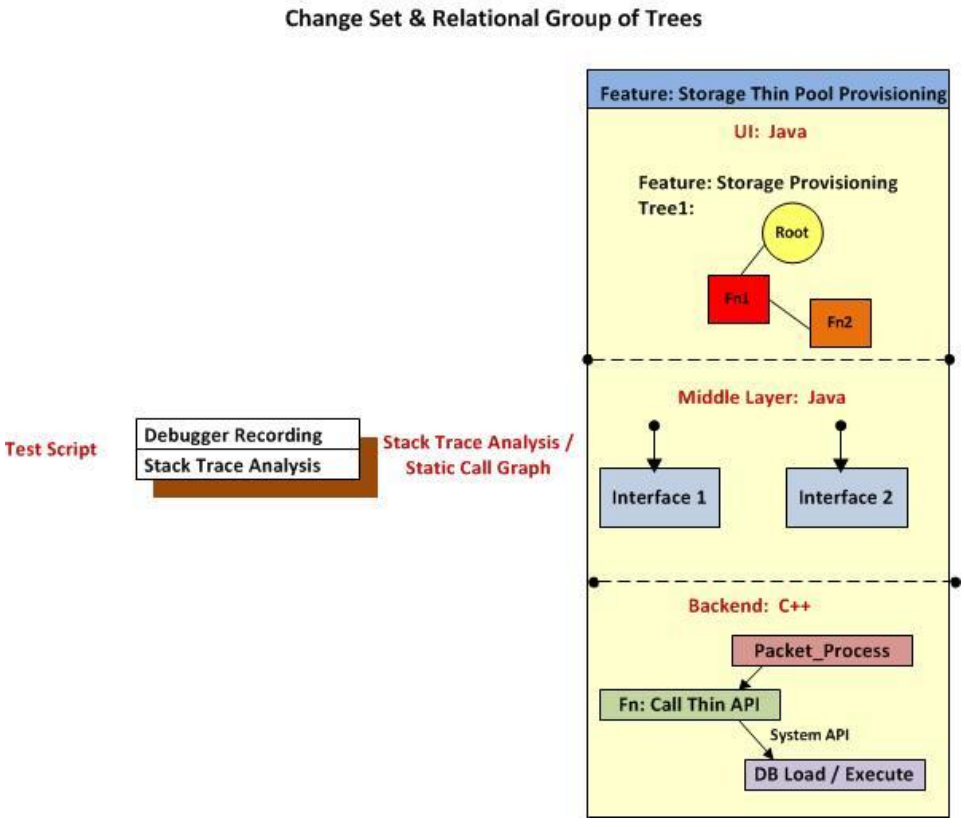
In this section we introduce the concept of a change set, which forms the core of the SAS system. The change set is a part of structural DB tagged and extracted by feature. All the feature tags corresponding to a feature are tagged under a sequence. The change set is a hierarchical grouping of tags related to a feature. The change set can be formed from the output of different types of testing (white box/black box testing).

The change set refers to the code and the relationships between the code of different components, applicable to a particular feature. Normally the change set information is obtained from a configuration manage system only while a feature is developed. In this paper we are able to derive the change set for a feature from the structural database as well as runtime recordings.

The change set internally can be referred to as a hierarchical group of relational trees derived from the structural database. The change set can contain sub-change sets corresponding to sub feature test cases. The sub-change sets are merged and augmented incrementally to form the feature change set which is a group of trees at each layer of the multi-tier software (as illustrated in the later diagram).

The rationale behind the concept of a change set is to provide a semantic meaning to the paths of the structural DB. When a test case is executed, the test case is tagged to the path traversed in the structural graph (which is stored in a structural DB). This forms the central idea of this paper. Once we have these tagged paths, we can process them, extract information from them, and build overlay applications from them (such as parallelized testing and automated root-cause analysis as discussed later). The next section and accompanying diagram show how a change set is generated from the structural database and the test case execution.

We have considered a multi-tiered application and a use case feature that is being tested. This usecase encompasses logic from the different tiers of the system like UI, Middleware and Back-end layer. As we can see in this example such a feature can be implemented using different programming systems but the heterogeneous data can be stored in a database.



5. Relational Extractor DB

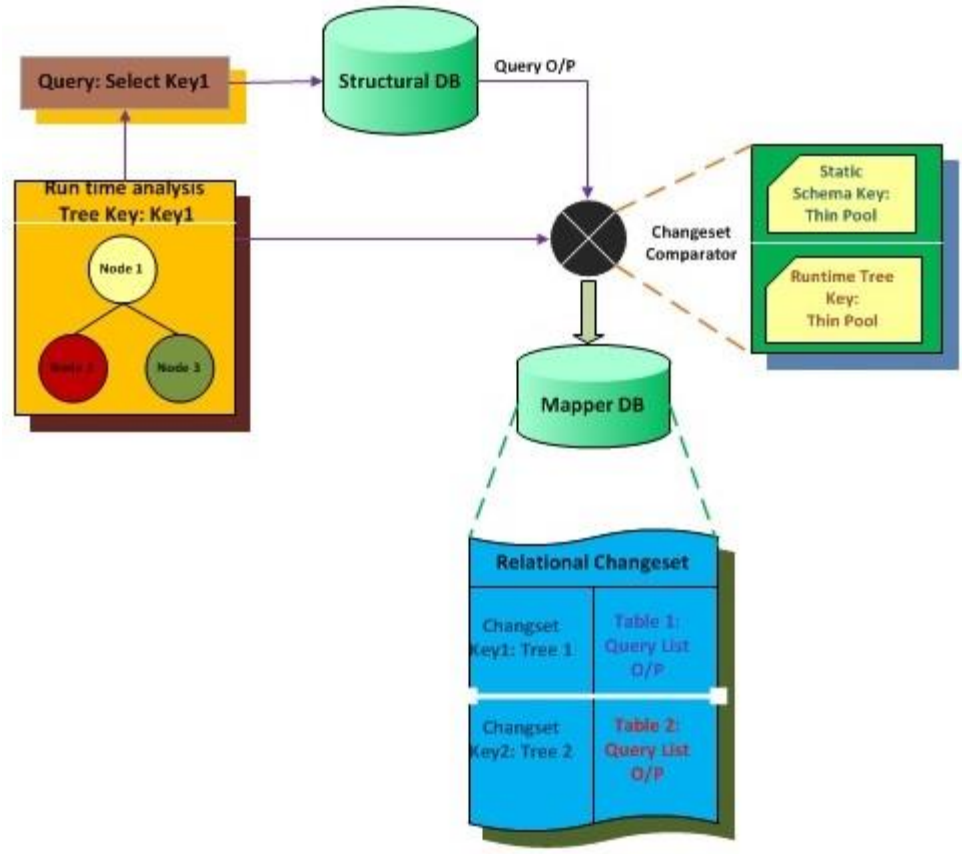
The purpose of the Relational Extractor DB is to merge the structural database content and the runtime feature based content. This feature-based content comes from the testing and the corresponding runtime content recordings.

The merge operation of the Mapper and Relational Extractor works by means by identifying paths obtained from dynamic analysis and, matching them with paths of the static analysis output i.e. the structural call graph DB. The matched path of the structural DB across all layers is extracted from the DB, decorated with a tag name (feature name) and added to the change set which is present in the Relational extractor DB. The output of the merge operation is a change set for a feature. This change set operation is ‘augment change set’ as the change set for a feature gets augmented as more and more paths are added to it, as more and more testing of the software takes place.

The querying capabilities of the system are dependent on this merge operation. As is discussed in the next section, the overlay applications are based on the Mapper DB created after the merge operation.

The figure below shows how the merge operation is performed from the operands of structural database and feature recordings. The merge operation can also be called as tagging, as it essentially assigned the feature tags to the corresponding paths of the structural database. The output of the operation is a relational extractor DB which stores the change set which has been defined in the previous section.

The test case maps to paths in the call graph database, known as the structural DB.



An aspect of the maturing DB is the change set expansion as the feature is covered more and more in testing.

Tagging of Structural DB (also known as the Merge Operation):

The change set is a part of structural DB tagged and extracted by feature. All the feature tags corresponding to a feature are tagged under one sequence. The change set is a hierarchical grouping of tags related to a feature. The change set can be formed from the output of different types of testing (white box/black box testing)

The aggregation of paths with tagged data but with gaps is a candidate for end to end path to be tested. This leads to a new test case not covered before in the different forms of testing in the system.

The concept of tagged paths is how the relational change set is created from the static and runtime analysis output.

The flow of the system is as follows:

Tagged Paths -> Categorization -> comparison based on testing tag.

The pseudo-code for the implementation of the merger operation is as follows:

```
Merge<dynamic_analysis_graph_1, call_graph_2>
 {
     Loop {
          Search for Keys<Path1 of graph_1, call-graph>
          Path-compare and extract compared path;
          Add to Changeset_1
     }
     Returns Changeset_1
}
```

The numbering scheme for the tagged paths is similar to that of below:

Testing_type[i].Majorno[i].Minorno[i]

This kind of numbering scheme enables user querying of tagged path. The Relational Extractor DB stores the test case table with the tag numbering scheme associated with the test case. The numbering scheme is derived from the type of testing, test case id and the feature and sub-feature wise classification of test cases. The test case table is as follows:

Test case Table:

Test Case Table: **Testing Type**	**Major number**	**Minor Number**
Unit		
White-box		
Black-box		

The aggregation of paths into a change set forest is depicted in the following table of change sets:

Feature ID	**Change set ID**	**Path IDs**	**Forest ID**
		List	

The Change Set implementation:

The change set for a feature is an expanding tree which gets augmented with more and more testing done and more and more tagged paths identified.

So the change set implementation should support the augment operations on the change set already created.

The pseudo-code for change set implementation is as follows:

```
Class Changeset

{

    List <Tree*> lt;

    Add()

    Modify()

    Merge(this, changeset1);

};
```

The high level pseudo code for merging a change set with its potential child change set is as follows:

```
Changeset::merge(this, Changeset1)

{

    For(;;) {

        Parent = this;

        If(this->tag.feature_name == changeset1->tag.feature_name) {

            Parent->addChild(changeset1);

            Break;

        }

        Else {

            This=this->next;

        }

        }

}//End of Changeset::merge() operation

Class Changeset_Track : public Changeset
```

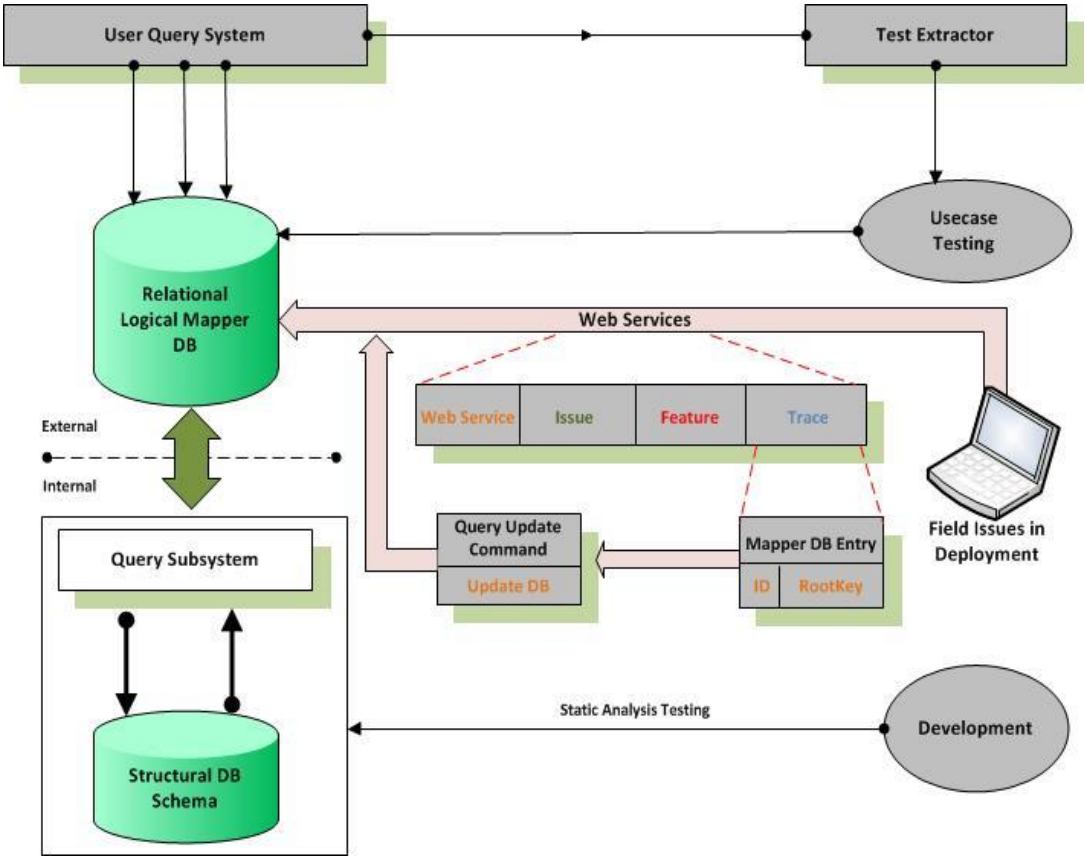
To summarize, the change set in the Relational Extractor DB maps the higher level test cases to the lower level paths. In other words the change set DB captures the path traversed in the structural DB for a feature test case or a group of test case for a feature. The change set of the Relational Extractor DB is extracted from structural DB for the paths matching the dynamic analysis, tagged as feature paths. There is a version match between the change set DB and the structural DB. The change set DB gets augmented in a loop-based fashion as more and more of a feature is tested. The change set DB stores the change sets which are hierarchical in nature, because they correspond to the higher level test cases.

6. Overall System

The overall system closes the loop between testing and feature development/fixes for issues. The database-based system keeps maturing as the data is fed into it and gets stored in a relational way. This enhances the capabilities of the user query system.

The interface between the field deployment of the software and the mapper DB is in terms of web services APIs. The issue (bug identified in the field) is mapped to the corresponding feature of the software and its identification in the mapper DB. This enables the updating of the DB using the corresponding query update command.

The overall system is described in the diagram below. The fixes made to the software are reflected in the overall system as the updating to the structural DB and the corresponding updating of the relational mapper DB. This ensures that the overall system works in a cycle leading to maturing of the database over time- which helps both in the software development / testing cycle and in post-release maintenance.



To summarize, the central idea of the SAS system is to deconstruct the structural DB by means of test cases (and the corresponding change sets) and assign meaning by assigning user defined, test case defined tags to paths in the individual paths of the call graph. In other words, the paths of the call graph, structural DB is extracted by its correspondence with the test cases and decorated with the test case keys namely tags. Then it can be constructed back by merging the change sets incrementally so that our goal of maximum coverage of the original call graph is achieved.

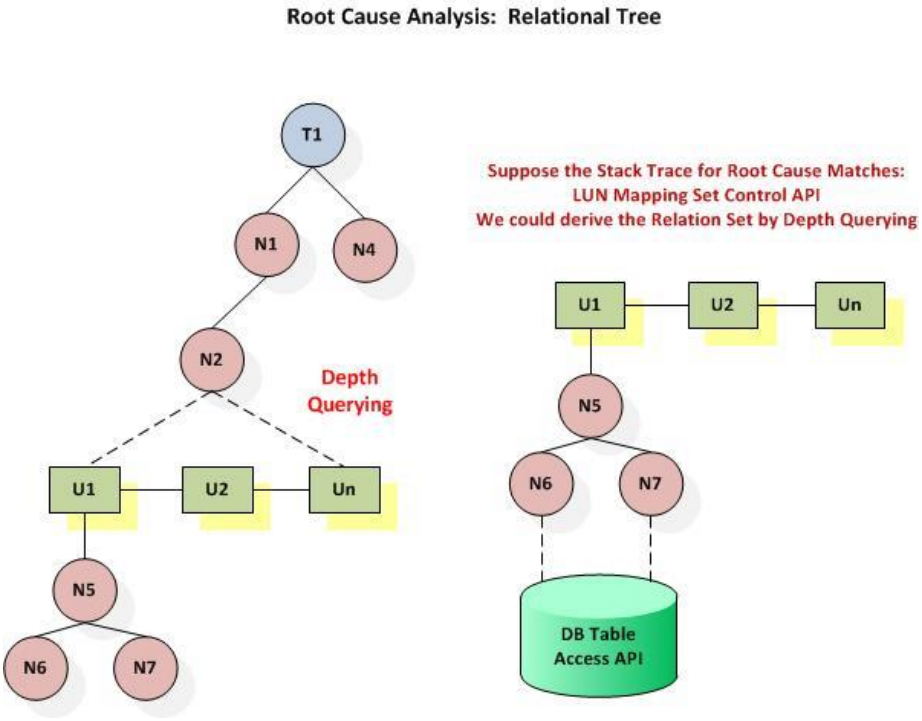
7. Case Study

The SAS is designed as a minimal system, which provides the user querying capabilities over the vast depth of software knowledge. This system enables developing simple queries, which apply this knowledge to the problems of software testing and maintenance. This section provides two such applications which helps in root cause analysis and parallel testing

7.1. Overlay application Relational Tree Visualization by Depth Querying for Root Cause Analysis

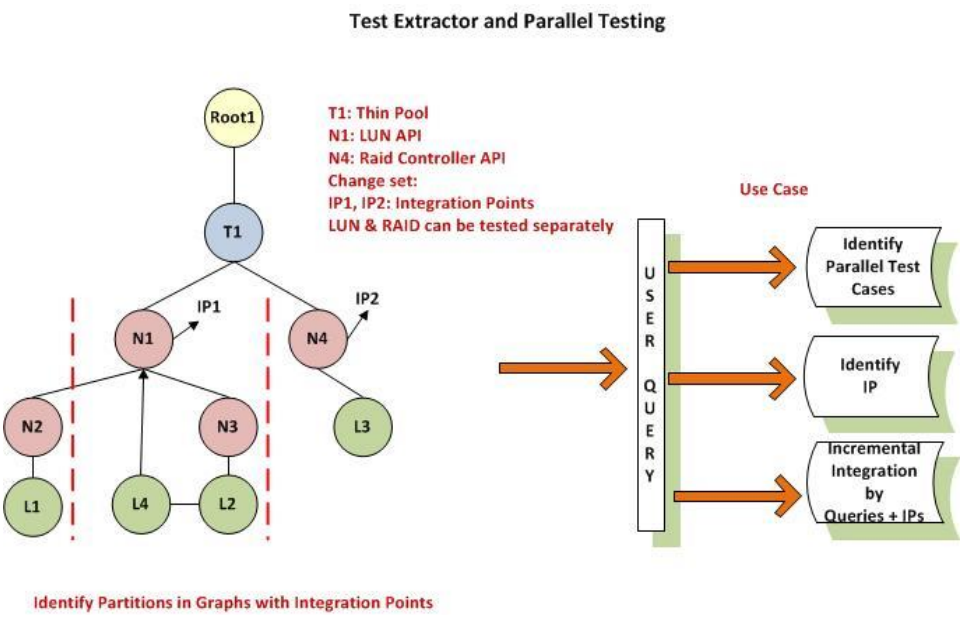
The advantages of the SAS system relational tree visualization over the available visualization of existing static analysis software (like (Doxygen)) is that it provides a heterogeneous change set which can aid root cause analysis over a multi-tier software application.

In the figure below from the stack traces of a bug, we could identify the corresponding test case affected by the bug. And by querying the SAS database we could identify the visualization of paths affected by it using Depth Querying across the layers of a multi-tier application. In the figure below, U1, U2 and U3 refers to the starting points of interfaces in a different layer for an end-to-end path scenario.



7.2. Parallel Graph Search in relational tree for Parallel Testing

One of the frequent problems of software projects is that feature completion of development is a dependency for software testing. By parallel testing, the aim is to perform parallel unit testing when the software is being developed and incremental integration testing.



In the figure we introduce the concept of integration points. An integration point is the point in the call graph where an independent logic branches out. Integration Points help in identifying parallel test cases.

The integration point can be thought of as the point up to which duplication/common paths are allowed in two potential test cases of the feature. This is to avoid false negatives in identifying parallel test cases caused by various factors such as common driver code, or code common to all paths such as common database code, or common networking library code.

The application queries the SAS system and identifies the partitions in graphs with integration points. The procedure to identify the test cases is to identify maximal length paths in the relational graph with no duplicates.

The following is an overview of how Parallel Testing test cases can be derived from the change set functionality of the Relational Extractor DB. The API accessed is the Identify_Parallel() API for the parallel test cases.

The pseudo-code for the parallel test case functionality is as follows:

```
Identify_Parallel(FeatureID [IN], ChangesetID [IN], PathsList(paths) [OUT])

{

    //Iterator over the PathsList

    If(PathsList != NULL)

    While( i <= PathsList.length())

    {

        addTestCase[PathsList[i]];

    }

}
```

The API for Test Executer is: TestExecute<TestCaseID, <ChangesetID,FeatureID combo>>

The logic for Test Tracker is to query the DB and the above said process of tagged path creation, updating of the change set for the feature happens again for this cycle.

New Test cases (Missing Gaps of Testing):

A query of the uncovered path (that is paths that are not tagged by any previous testing) in the call graph will yield data for the new test cases to be designed.

Examples

1. SAS - Parallel Regression Automation (Feature development)

This example scenario is to manage a storage software system which is made of a RAID group of disks and has advanced storage provisioning features like that of the storage pool. Here is some information on the software being developed in an agile scrum model.

- RAID group Feature
- Virtual Provisioning Feature
- Storage pool feature

Each feature is built in incremental iterations of the product development process using the agile scrum model.

The key feature of this model is that there is a demonstrable, testable deliverable of software development at the end of each iteration of the project.

RAID Group feature can be classified into two categories:

1. Discovery of the RAID Group component and its sub- components.

2. Active management of the RAID Group feature which performs an action on the existing RAID Group in the system, thereby changing its configuration. After the active management operation a discovery of the changed RAID Group component is needed to reflect the changed component in the User Interface.

Now consider the virtual capacity calculation problem: It is a function of the RAID Group capacity calculation. There is a dependency here.

Now the inclusion of the additional features means the following changes to the implementation (code portion) of the software:

a) New members being added to the discovery object, updating of existing members of the discovery object.

b) New members being added to the database tables, new addition to the record set retrieval functionality and the corresponding changes in the User Interface.

Now consider the changes to the capacity calculation logic:

In which way new functionality affects the previous iteration existing code.

Previous:

Total Capacity = ∑ (RAID Group capacities of all the RAID Groups in the system)

Now:

Total Capacity = ∑ (RAID Capacity) + ∑ (Storage Pool Capacity) + ∑(Meta Capacity)

The representation in the Relational Extractor DB for the Storage Pool scenario:

- End to end paths tag, testing type: end-to-end
- Unit testing path tag, testing type: unit-test

Change set testing path1: StoragePool

1. End to end path tag – storage pool discovery,
2. End to end path tag- storage pool active management,
3. End to end path tag within active management :
    a. Create storage pool,
    b. Associate storage pool to RAID group,
    c. Storage pool expand.

All the three above active management paths (3.a-c) belong to the active management change set. The active management change set along with storage pool discovery path forms part of the storage pool change set.

The changes sets are hierarchical in nature, representing the structure of test cases in the testing domain.

The example of tags for the above three examples (1-3) are as follows. In the example E1 ..E7 are function calls which are part of the change set. For instance E1-E2-E3 is one such path of function calls traversed by the end_to_end.storagepool.discovery change set.

1. End_to_end.storagepool.discovery: E1 - E2 - E3
2. End_to_end.storagepool.active.create_storagepool : E1 - E4 - E5
3. End_to_end.storagepool.active.associate_storagepool_to_RAIDgroup: E1 - E6 - E7

In the above change sets (1) and (2) have E1 in common so it's a common interface method traversed by both the test cases i.e. storage pool discovery and storage pool active management operation: create storagepool.

The representation of the above change sets in the database table is as below:

Testing Type	Change Set ID	Feature Level1	Feature Level 2	Feature Level 3	Call graph function calls in path for this change set	Child Change set IDs
End to end	CS1	StoragePool				CS2, CS3
End to end	CS2	StoragePool	Discovery		E1→E2→E3	
End to end	CS3	StoragePool	Active			CS4, CS5
End to end	CS4	StoragePool	Active	Create StoragePool	E1→E4→E5	
End to end	CS5	StoragePool	Active	Associate StoragePool to Raidgroup	E1→E6→E7	

As can be seen from the database table, change sets CS1, CS2 and CS3 are related by the hierarchical relationship CS1→CS2, CS1→CS3. And CS3 is the parent change set for the child change sets CS4 and CS5 represented by the relationships, CS3→CS4, CS3→CS5. The above change sets are an example of hierarchical change sets.

Storage Pool Feature Testing:

Find parallel test cases in storage pool change set:

Query for paths in the changeset.storagepool.

Identify Integration Points: E1

Independent paths which can be run in parallel: E2-E3, E4-E5, E6-E7.

These independent paths that can be run in parallel are identified by the procedure provided in Case Study of Section 7.2. Basically, it is achieved by taking the integration point (E1 in this case) and identifying parallel paths in the graph of related change sets. From the graph we can deduce that beyond the Integration Point- E1, the paths in question E2-E3, E4-E5, E6-E7 are all independent paths with no common function calls. Therefore they can be run in parallel.

The corresponding test cases for the above paths:

- storagepool.discovery,
- storagepool.active.create_storagepool,
- storagepool.active.associate_storagepool_to_RAIDgroup

These change sets and their subsets can be tested in parallel.

New Test cases(Missed Gaps):

Query for Paths in Structural DB with unit testing paths in each layers but no continuous paths in end-to-end testing paths:

The query result will have such paths that have not been tested in the end-to-end scenario.

1. So proceed to test one of these end-to-end untested paths,
2. Rerun the merge operation of relational extractor db, this tested path becomes a tagged path.
3. The tagged path based on the tag detail gets aggregated to the corresponding change set.
4. This tagged path shows up in the query for parallel test cases. And if it is an independent path, the test case can be regressed as a parallel test case.

The high-level use cases are:

1. Identify module-by-module testing dependency.
2. Carry over testing from iteration to next. In integration testing, lower level layer functionality is tested with a stub. If the stub is replaced by actual higher layer, with SAS we are able to identify the testing dependencies. This is possible due to the longest subsequence match of the change set which has been affected from one iteration to next. The regression test can be calculated in the current iteration and testing is performed efficiently based on it.

2. Bug Scenario

The logic for bug is similar to parallel regression for feature.

Example Defect:- Capacity column of UI – Erroneous value due to missing virtual capacity.

Change set contents:

Back end:

Before Defect fix: the path flow in the change set:

StorageArray.Insert(Array_Group.capacity[IN], float x [OUT]) →

AClass.populate(StorageArray) →

StorageArray.setCapacity(float xval) →

Normalized_collected_capacity(RAIDGroupCapacity, xval) →

CollectCapacity(RAIDGroupCapacity) →

Parse_CommandExecuted(output,command) →

ExecuteCLICommand(“getrg –capacity”) → (1)

After Defect Fix (Changed portions/ new portions in Bold):

StorageArray.Insert(Array_Group.capacity[IN], float x [OUT]) →

AClass.populate(StorageArray) →

StorageArray.setCapacity(float xval) →

Normalized_collected_capacity(RAIDGroupCapacity + VirtualPoolCapacity, xval) →

CollectCapacity(RAIDGroupCapacity) → **CollectCapacity(VirtualPoolCapacity) →**

Parse_CommandExecuted(output,command) →

ExecuteCLICommand("getrg –capacity") → **ExecuteCLICommand("getvpool –capacity")** → (2)

The contents above in path (2) in bold show the additional changed paths in the call graph and in the change set after the defect fix.

Now the contents of the change set<Storage Array> before and after the defect fix are :

Before:

Tag<GetCapacity> ➔ (1)

Tag<Show Connectivity & Capacity> ➔ (1) → Tag<list of RAIDGroups>

Change set after the defect fix:

Tag<GetCapacity> ➔ (2)

Tag<Show Connectivity> ➔ (2) → Tag<list of RAIDGroups> ->Tag<list of VPools>

The above shows that apart from GetCapacity test cases, the test cases that need to be regressed for the specific bug fix are

"Show Connectivity" which in turn contains the test case for:

a) List of RAID groups

b) List of Virtual Pools

This is deduced from the path graph queried from the change set for the feature, Storage Array.

The concept of tagged path and change set also narrows down the combinatorial explosion of paths returned from the query of the structural DB.

Since the black box test case is tracked with the actual path traversed in the code, the accuracy of the method in regression identification is possible.

In the above example, Tag<Show Connectivity> change set and Tag<GetCapacity> change sets are the sub-change sets of the Tag<StorageArray> change set.

Like this there could be <StorageTapes>, <DiskLibrary> and <Router> change sets modeled after the feature test cases in the product.

Defect Regression:

> Related (connected) paths changed paths in the change set + paths not covered in the changed portion of the latest call graph (code). This is an example of new test cases identified by the SAS system.

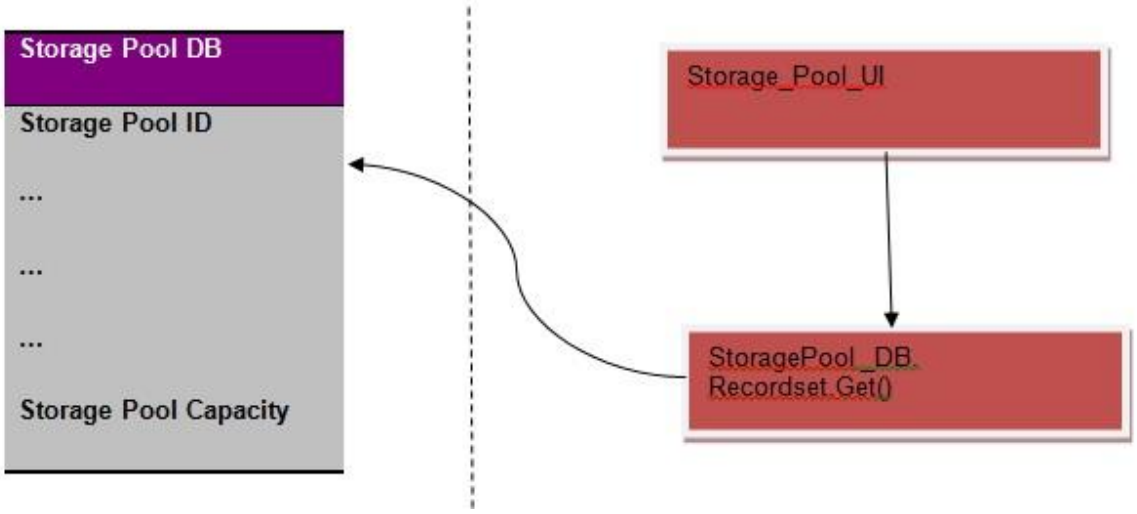
Consider a case where there are N number of defect fixes given in a patch. Then the group of defects can be regressed as an incrementally developed change set. This is the advantage of the hierarchically organized change set in the SAS system.

3. Another Bug Trace through the SAS tool

Bug 2: Negative value in capacity field of storage pool attribute

Query the Extractor DB for Storage Pool Feature.

- Identify the Change Set – for UI, DB and Back-end layer for the storage pool feature. The output will be something like this:



Changeset ID: Array.StoragePool

PopulateDB(StoragePool, StoragePoolID, float Capacity)

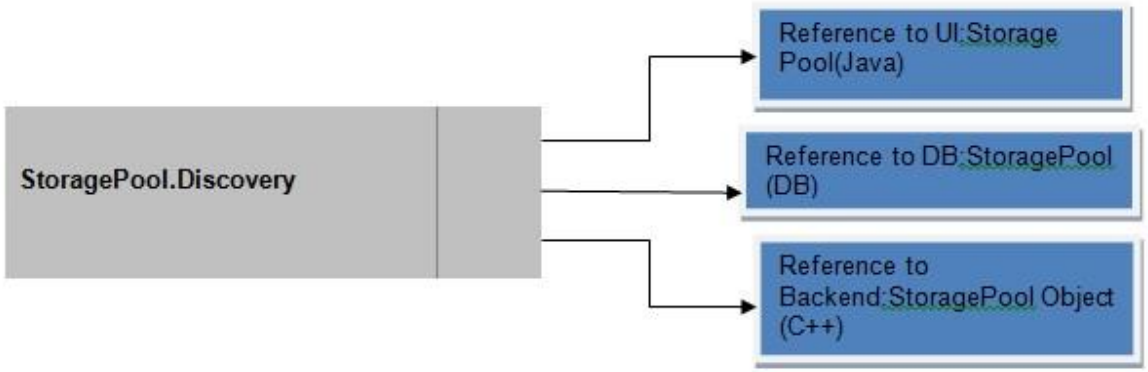
Change Set is a list of references to the call graph (represented in DB) queried at each layer of the software for each feature.

Query:

Get the information about “StoragePool.Capacity” in StoragePool.Discovery change set

By discovery we mean how the Storage Pool data found its way in to the UI field.

ChangeSet --------- List



Hierarchical queries will fetch the following information:

UI:

StoragePool_UI.field

StoragePool_DB.recordset.Get(Capacity)

Query:

DB.StoragePool(StoragePool_DB, capacity)

The query output of the above query will lead to the following functionality in the Back End (C++ layer):

StoragePool.insert(StoragePool.AClass.capacity, float x);

[AClass is a structure in which data is sent from the C++ Backend layer]

SAS Query: Get the info of StoragePool.AClass(capacity) =>

Aclass.Populate(StoragePool) =>

Aclass.StoragePool.SetCapacity(float xVal) =>

SAS Query for xVal in Backend Layer:

Yields normalize([in]arrayCapacity, [out] xVal) => This is the function where the bug lies.

Regression:

Execute Test => Stack Recordings

Compare Path => Verify coverage takes place (minimal set regression)

This enables rigorous testing during each iteration of development before the product gets shipped.

Here the advantage of SAS concept of change set is that because it tracks the changes across the layers we know the changes corresponding to a feature/bug in totality.

A group of change sets for the corresponding features can be provided to testers to test based on coverage. Based on the change set coverage in the overall call graph structural DB, we know the paths covered, yet to be covered, identify parallel automatable test cases and uncover new defects.

How SAS tool helps in testing during product development:

SAS-tool based actual dependencies of test cases on the basis of real code of the software are more accurate than perceived dependencies of the test cases.

Bug: Now the fix for the bug is related to the field size of the data member leading to overflow error. This fix needs to be made at all the layers of the software (UI, middle DB, Backend C++). The utility of SAS is that the call-graph changes for this fix is stored in a change set (remember the fix is at all layers, so the change set is a list of changes at each layer). This storage of change set can be retrieved quickly for further queries thereby leading to a system which keeps improving.

Regression Testing:

Once the bug fix is made,

Execute Test => Stack Recordings

We can query the DB for paths relative to this path, if it is an independent path or else it's a dependent path. All the dependent paths are extracted from the tags we can identify the corresponding test cases and they need to be rerun for minimal set regression.

The use of SAS here is not just for regression of a single bug but dependencies between bug fixes (testing of incremental patches that have been provided). This helps the tester tremendously.

8. Applications of SAS

The applications of SAS can be summarized as follows:

1. Identify Parallel test cases in test case regression of a feature/group of related features.

   The interesting aspect of this kind of parallel testing is it easily applies to the Agile model of software development, which calls for a demonstrable partial product of the overall product in each iteration cycle (sometimes called a sprint) of the software development process.

   The following questions can be answered by the SAS tool:-

   - How could testing take place?
   - What are the regression implications as more and more features are added to the product?
   - How to we identify opportunities for parallelism?

   Another use of the SAS tool is the test case categorization in terms of identifying the embedded test cases and those which are not.

2. Bugs and Root Cause Analysis.

An example for this application of SAS tool is given in the case study 1.

3. Bugs and Regression

   A bug fix is made with the help of SAS RCA analysis. The structural DB is updated after the bug fix. Changes set for bug is calculated – Minimal set regression of the bug across a multi-tier application is possible in the SAS tool.

4. Test Execution Tracking (based on comparator and Structural DB)

   The SAS tool helps in automatically identifying test cases that were not covered in one level of testing- say earlier iteration or unit testing. This helps in identifying missing pieces in development testing.

Logical to Physical Mapping of test cases helps in:

1. Minimization of regressions (feature).
2. Dependency – Visual dependency of test cases - an application of which is parallel test cases.
   Visualization of test cases -
   a) An organization of test cases that is from ad-hoc organization to actual dependency based organization of test cases. It helps tremendously in testing optimization and automation.
3. As we model every static and run time object, it helps in root-cause analysis as well.
4. Minimization of bug regression.
5. Complex end-to-end test cases can be easily designed and validated. New test cases can be validated before execution itself by querying the SAS database.

9. Effectiveness of this approach:

1. Visualized Testing – it merges the benefits of white box/black box testing

2. Agile Process – Avoid delayed testing - SAS allows early and effective testing, SAS helps in integration testing and end-to-end testing.

3. More confidence in the quality of the product – visualized tracking of testing effectiveness. Metrics can be derived about the testing coverage as a proportion of the call graph DB.

   An important metric for testing using SAS is: Overall Project Testing effectiveness in terms of coverage. This can measured in SAS by means of cumulative change sets for a feature. This cumulative merged change set should be as close as possible to the Structural DB. The percentage of the match between the two gives the percentage of the overall project testing effectiveness

The effectiveness of this approach is that Queries of the SAS system can be cached and result set retrieval optimization increases the efficiency of the system.

The other aspect of its effectiveness in the maturing database is the fact that there could be many instances of the same bug found by many customers. All of them have similar RCA and bug regression, such instances can easily be tracked and handled by the SAS system.

The new metrics of testing possible due to SAS is the analysis of nature of bugs, inter-dependency of paths in the call graph structural DB leading to identification of inter-dependence of modules from the testing perspective. There has been work related to dynamic analysis for identifying the code impact (Law 2003) and also identify test cases (Frechette 2013), but they do not cover a complex multi-tier scenario and also the host of advantages of the SAS tool listed above. The storage based approach helps in

internal vulnerability assessment leading to new test cases and possible new types of bugs in the multi-tier software system.

10. Conclusion

The objective of this paper is to provide a glimpse of what can be done with software analytics based on structural analysis and storage. For this purpose the SAS tool has been designed to work on call-graphs of the software at different architecture levels, by storing it in a database and processing it. In this we have specifically shown how the logical, user domain organization of test cases is mapped to physical path organization of the test cases based on the call graph and storage based approach. This enables a host of applications from developers to testers from Root-Cause Analysis to test case design, execution, tracking and measuring overall effectiveness of the testing.

References

Callahan, D.; Carle, A.; Hall, M.W.; Kennedy, K., "Constructing the procedure call multigraph," Software Engineering, IEEE Transactions on, vol.16, no.4pp.483-487, Apr 1990.

Frechette N, Badri, L, Badri, M, Regression Test Reduction for Object-Oriented Software: A Control Call Graph Based Technique and Associated Tool. In the proceedings of ISRN Software Engineering, March 2013.

Gerter O. 2004. A Database File System. An Alternative to Hierarchy Based File Systems. University of Twente.

Graham, Dorothy. The Foundations of Software Testing. Wiley Publications.

Grove, D., DeFouw, G., Dean, J., and Chambers, C. 1997. Call graph construction in object-oriented languages. SIGPLAN Not. 32, 10 (Oct. 1997), 108-124.

Law, J, Rothermel,G. "Whole program path-based dynamic impact analysis," in *Proceedings of the 25th International Conference on Software Engineering*, pp. 308–318, May 2003.

Oxygen Static Analysis Tool for C, C++, Java, Python, etc.(http://www.stack.nl/~dimitri/doxygen/)

Patent: Method of implementing an acyclic directed graph structure using a relational data-base http://www.google.com/patents/US6633886

Ryder, B.G., "Constructing the Call Graph of a Program," Software Engineering, IEEE Transactions on, vol. SE-5, no.3pp. 216- 226, May 1979.

Stack Trace Analysis. https://profes2013.cs.ucy.ac.cy/files/GavrilovStackTraceAnalysis.pdf