Augur: Incorporating Hidden Dependencies and Variable Granularity in Change Impact Analysis

Tushar Sharma
Dept. of Management Science and Technology
Athens University of Economics and Business
Athens, Greece.
tushar@aueb.gr

Girish Suryanarayana
Corporate Research & Technologies Center
Siemens Technology and Services Pvt. Ltd
Bangalore, India.
girish.suryanarayana@siemens.com

***Abstract*—Software change impact analysis (CIA) methods enable developers to understand potential impacts of a code change so that the change can be executed confidently without affecting reliability of the software. However, existing CIA approaches do not support CIA for all source code granularities. Additionally, they lack support for inter-granular change impact queries and hidden dependencies. This paper introduces Augur, an automated static code analysis-based CIA approach that addresses these shortcomings. Augur infers and maintains semantic and environment dependencies along with data and control dependencies between source code entities across granularities. Additionally, Augur uses Change Impact Query Language, a novel query language for impact analysis proposed in this paper, to support inter-granular CIA queries with batch querying feature. Augur has been realized as a Visual Studio extension called Augur-Tool. We have conducted quantitative evaluation on two open-source and two industrial projects to assess the accuracy of the tool. Results from the evaluation indicate that Augur provides CIA with high accuracy (average precision 55% and average recall 85%).**

***Keywords*—Change Impact Analysis, Change Impact Query Language, Semantic dependency, Environment dependency.**

I. Introduction

The software maintenance phase is the longest phase of the software development lifecycle that spans 40-80% of total project effort and cost [1]. Developers carry out adaptive, corrective, perfective, and preventive activities during the maintenance phase to constantly improve the software and deliver value to their stakeholders. Each of these activities introduces a set of changes in the software system. In order to make a change confidently without affecting the reliability of the software (i.e. making a change without introducing new bugs), a developer has to be aware of potential impacts of the proposed change. A recent survey has identified unknown impacts as the biggest technical deterrent for carrying out refactoring [2].

Therefore, the topic "Change Impact Analysis" (CIA) is receiving increasing attention from the software engineering community. CIA is defined as *"The determination of potential effects to a subject system resulting from a proposed software change"* [3] and aims to minimize the unknown and unexpected side effects of a change.

Foreseeing the impacts of a proposed change is difficult; the amplitude of the difficulty increases as the software system becomes larger and complex. Even in the case of an experienced developer who has spent considerable time with the software, it is not guaranteed to cover all potential impacts of a change just by following his understanding about the software.

Researchers have, therefore, focused on automated methods for CIA. One set of automated CIA methods uses historical information from version repositories to mine co-change dependencies (i.e., eliciting the source code entities that are changing together) and analyze the impact of a future change based on the co-change history [4]–[12]. However, a CIA method based on mining history depends on the past operations and existing dependencies. Therefore, it might lead to incorrect or incomplete results especially when new artifacts or new dependencies are introduced in the software. Further, most of the methods belonging to this category treat a repository check-in as a transaction and hence assume that all the changed source code entities since last check-in belong to a single related change which is often not the case. Such transactions introduce noise in the CIA and lead to incorrect results.

Many other CIA automated methods use static code analysis to determine the impact of a proposed change [13]–[19]. However, they have the following shortcomings: First, most of these methods work on a single granularity of the source code (either class or method-level) and do not support CIA for all source code granularities (i.e., local variable, field, statement, method, class, and namespace). Developers prefer impact results at multiple granularities to better understand the impact of a change. Thus, a single granularity for CIA is not sufficient and leads to imprecise results. On the contrary, allowing developers to choose variable granularity improves precision and provides more useable and actionable change impact information [19].

Second, these methods do not support inter-granular queries for CIA. An inter-granular query specifies the proposed change at one source code granularity and provides the result of the query at another source code granularity. For example, the query "Show the methods that possibly need to be updated (across classes) when I make a change in this class" is an inter-granular query. In contrast, most of the existing approaches would have only informed what other classes would be impacted by the proposed change. Support for inter-granular queries in a CIA tool can precisely pinpoint the impact of

a change leading to a more effective CIA.

Third, existing static analysis-based CIA methods only use data and/or control dependencies to identify the impacted source code entities. It has been shown that recall of such methods is low due to the presence of hidden dependencies [20]. While there has been some initial work towards identifying these hidden dependencies [20], such dependencies have not been properly explored and are not supported by existing CIA methods.

Fourth, activities such as refactoring impact multiple source code entities within a single refactoring step. However, existing CIA approaches require the developer to manually invoke change impact analysis for each of the changed source code entity. This is a cumbersome task since the developer has to himself keep track of the change impact with respect to each of the code change steps and then aggregate it to reason about whether he should proceed with the refactoring. It would be helpful for developers to have a tool that allows them to club all the steps within a refactoring and invoke a "batch" CIA on those steps to receive an overall summary of the proposed changes. However, our survey of existing CIA approaches shows a lack of support for this feature.

To address these shortcomings, we propose Augur, a static code analysis-based CIA approach and realize it as a Visual Studio extension which supports

- change impact analysis at all source code granularities
- inter-granular CIA queries
- semantic and environment dependencies along with data and control dependencies
- batch CIA using a novel Change Impact Query Language (CIQL).

II. AUGUR

This section describes how Augur addresses the shortcomings of existing CIA methods described in the previous section.

A. Dependencies supported by Augur

Traditionally, data and control dependencies have been employed in CIA; however, there are hidden dependencies such as semantic and environment dependencies identifying which helps improve the accuracy of CIA. We discuss below the various dependencies that are supported by Augur. In the following discussion, A and B refer to source code entities which could either be local-variables, fields, statements, methods, classes, or namespaces.

1) Data dependency: If A uses B in its direct computation, then A is directly dependent on B. Data dependencies are transitive, thus if A uses B indirectly via intermediate dependent entities, then A is indirectly dependent on B.

2) Control dependency: If B is a part of a controlling or conditional expression of a control structure (such as an *if* statement) and A is a part of a statement within the control structure (such as *then* block of an *if* block) such that a change in the value of B decides the execution of A, then A is dependent on B (through control dependency).

3) Semantic dependency: Semantic dependency refers to dependencies among source code entities that arise due to the semantic relationships between programming language constructs. Currently, Augur supports the following types of semantic dependencies:

- When A and B are constructors of classes X and Y respectively (where Y is a sub-class of X) such that a change in A may impact the object creation of class Y through B, then B is semantically dependent on A.
- When A and B are overloaded methods in a class, a change in method signature of one of the methods may change the method invocation order involving A and B, then A and B are semantically dependent on each other.

4) Environment dependency: If A and B interact with each other via the environment such as using database, registry, or file, they share an environment dependency between them.

B. Unified Dependency Graph

The Unified Dependency Graph (UDG) is a novel construct that we have developed for Augur to conveniently and effectively capture and represent source code entities and their relationships. UDG is a conceptual extension of DSD (Data and Structure Dependency) graph [21]; it infers and maintains the dependencies supported by Augur for all source code granularities. UDG also maintains information about additional relationships such as inheritance and method calls between source code entities. UDG is modelled as a hierarchical graph where each plane of nodes (i.e. vertices) represents a source code granularity. UDG provides a set of APIs to access the contained information.

Figure 1 shows a UDG where a software system is represented by a node that has references to its children nodes that represent namespaces in the software system. Each namespace node contains a reference to a graph representing classes in that namespace. Each class node contains references to two graphs — a method and a field graph. Each method node contains references to two graphs — one represents all the statements and the other contains nodes representing all the local variables in the method. Similarly, each node also maintains the reference to its parent node.

Additionally, if a node has one or more dependencies on another node at the same level, it maintains a reference to it. Finally, the graph also maintains inheritance and method-call relationships. Class nodes keep references of other nodes that represent its sub or super class. Similarly, method nodes keep references of other nodes that represent methods called from it. Nodes representing fields, statements, and local-variables may keep references of other nodes in the same graph or in other sibling graphs on which they have one or more dependencies.

The dependencies among the source code entities are populated, computed, and maintained in a bottom-up fashion. Thus, if a dependency between two source code entities at a higher-level granularity is arising due to a dependency between entities at a lower-level granularity, then the UDG maintains the dependency only once between the nodes representing the original source of dependency. This avoids the risk of

inconsistent dependency edges in the graph due to duplication of dependencies. It is possible that two nodes share multiple edges due to multiple relationships or dependencies.

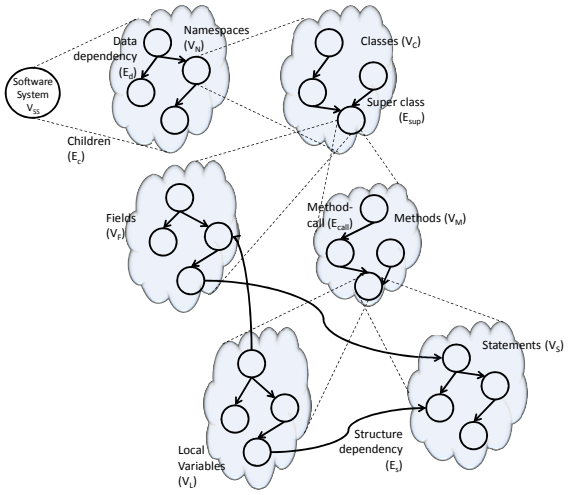

Fig. 1. Unified Dependency Graph

1) Supporting Inter-granular Queries: An inter-granular query specifies a proposed change at one granularity level and expects the corresponding potential impact set at a different granularity level. UDG plays a vital role in helping Augur support inter-granular queries. As described earlier, UDG is a hierarchical graph where each plane of nodes represents source code entities at different granularities. The hierarchical structure of the graph enables Augur to conveniently infer and maintain references to dependent source code entities for different granularities.

C. CIQL *(Change Impact Query Language)*

To support batch execution of multiple Change Impact Queries (CIQs) in a scalable and convenient way, we developed a new query language namely Change Impact Query Language (CIQL). The language helps a user to easily formulate CIQs by using a rich set of CIA options. The CIQL specifies a grammar and defines the syntax for queries. It supports the specification of a CIQ along the following dimensions:

Granularity (Source): Granularity of the source code entity intended to be changed. It can be one of the following: *Local variable, Statement, Field, Method, Class,* and *Namespace*.

Entity: Specific source code entity intended to be changed.

Analysis depth (optional): The depth of analysis expected from Augur. It can be assigned one of two values: *Direct* (default value) or *All*. A *Direct* depth specifies that the user is interested only in the direct potential impacts of the intended change, whereas an *All* depth states that the user is expecting to carry out direct as well as indirect impact analysis.

Scope (optional): The scope within the software system where the potential impacts will be looked for. A scope could be one of the following: *ContainingClass* (where the specific entity is residing), *Classes* <classes> (a set of classes specified by <classes>), *ContainingNamespace* (where the specific entity is residing), *Namespaces* <namespaces> (a set of namespaces specified by <namespaces>), or *SoftwareSystem* (the whole software system; this is the default option).

Granularity (Impact): Granularity of the potential impact set. It can be one of the following: *Local variable, Statement, Field, Method, Class,* and *Namespace*.

1) Syntax of queries: The CIQL enforces the following syntax for CIQs:

CIQL::get "<Granularity (Impact)>" [within "<Scope>"] [with "<Depth>"] where "<Entity>" is "<Granularity (Source)>".

Each CIQ starts with "CIQL::get" to explicitly state the protocol of the query, i.e. CIQL and the operation i.e. *get*. Any entry in angle brackets specifies the placeholder for the entry; for instance, <Entity> specifies the specific source code entity for which the change is intended. For instance, to specify a method, one has to specify the namespace, class of the method, and the method name separated by semi-colons. In case, the granularity specified is a *Statement*, then the statement number within the containing method needs to be provided. Optional entries are shown in square brackets.

D. Augur-Tool

We have realized Augur as a Microsoft Visual Studio Package [22] (called Augur-Tool). A Visual Studio Package is an extension that seamlessly integrates itself into the IDE and helps highlight the produced output (in this case, inferred potential impacts) in the IDE itself. The Augur-Tool supports .NET version 4.5 and can be installed on Visual Studio 2012, 2013, and 2015 editions. The tool along with instructions to install and use can be found here: http://bit.ly/Augur. Augur-Tool follows the architecture showed in Figure 2. The following sub-sections describe the components in the architecture.

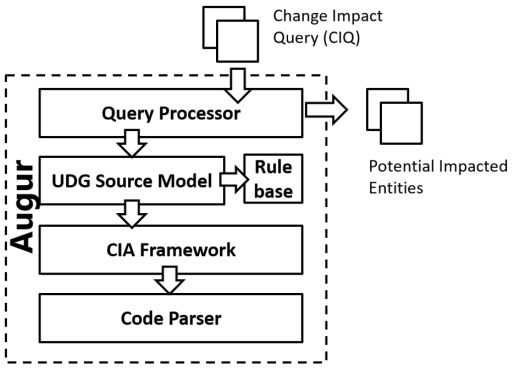

Fig. 2. Architecture of the proposed method

1) Query Processor: A user provides a CIQ as input to Augur via the UI as shown in Figure 3. The Query Processor is responsible for receiving the query, interpreting its various dimensions, and evaluating it using the source model provided by the UDG. Upon completion of the query evaluation, the results of the query i.e. potential impact sets are either relayed to the UI or logged to a file in case of batch queries. The UI

allows the user to click on any of the produced impacts and shows the corresponding source code entity in the IDE for easier navigation. Augur provides a separate UI to specify a CIQ using CIQL. This interface allows the user to specify either a query directly or a CIQL batch file containing multiple CIQL queries. The CIQL batch file offers a scalable way to compute change impact for a large number of proposed changes without manual interference.

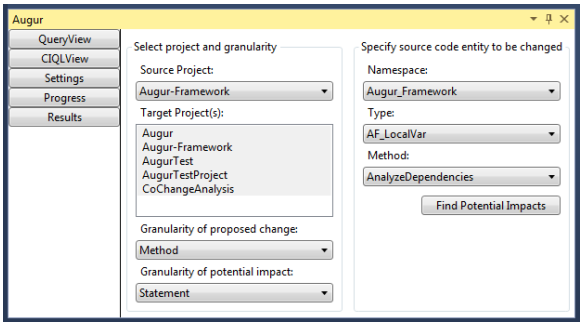

Fig. 3. QueryView in Augur-Tool to specify a CIQ

2) UDG *Source Model:* The responsibility of the UDG Source Model is to use the CIA Framework to infer relationships and dependencies among the source code entities and populate an instance of UDG (see Section II-B) with the inferred information. This information is used to evaluate queries sent by the Query Processor.

The UDG Source Model creates the UDG graph instance (i.e. it creates graphs at each granularity) at application start-up. Once an graph instance is created, the UDG Source Model applies a set of rules encapsulated in the Rule Base module. The rules are applied on the source code information in the CIA Framework to infer relationships and dependencies between nodes. The graph is then populated with this information.

3) Rule Base: The Rule Base module not only realizes rules to detect data and control dependencies among program entities via generic traditional implementation, but also implements a predefined set of rules to infer semantic and environment dependencies. Currently, Augur supports rules for semantic dependency arising from constructor dependency in a class hierarchy and overloaded methods. Similarly, environment dependency arising from shared access of file, database, or registry is supported.

It is important to note that these rules are specific in nature and typically cover one facet per rule. For example, where two source code entities share a dependency because of their interaction with a same registry key, our rule base provides a rule that checks for instances of the same registry key across the software system and infers environment dependency.

4) CIA Framework: The CIA Framework abstracts the low-level Abstract Syntax Tree (AST) library provided by the underlying Code Parser and provides a higher-level convenience API to the UDG Source Model. Thus, it separates the application specific logic from the source code parsing logic. Having such an explicit layer of abstraction allows the underlying parsing library to be replaced in the future (for instance, to support another programming language) without impacting the UDG Source Model layer.

5) Code Parser: The Code Parser component is responsible for parsing the source code and generating an AST to be used by the CIA Framework. For the Augur-Tool, we have used NRefactory (an open-source library) [23] as the Code Parser. NRefactory also provides a set of APIs to access the source code information programmatically via the generated AST.

III. Quantitative Evaluation

The goal of the quantitative evaluation was to assess whether and to what extent Augur improves the accuracy of CIA. Towards this, we picked two open-source projects and two proprietary industrial projects as the subject systems for the quantitative evaluation. The open-source projects are GitExtensions [24] and VsVim [25]. The industrial projects (referred as P1 and P2 in this paper) belong to Siemens CT DC AA (Corporate Development Center Asia Australia). All of the projects are implemented largely in C#. Characteristics of the subject systems are listed in Table I.

TABLE I
Characteristics of quantitative case-study systems

Software system	LOC	#Types
GitExtensions(version 2.48)	89K	499
VsVim(version 1.8.0.0)	88K	398
P1	58K	479
P2	72K	509

1) Evaluation process: To evaluate the accuracy of the Augur-Tool, we employed two widely accepted metrics namely precision and recall. For a given intended change, if M is the correct impact set and A is the estimated impact set, then the precision P is the fraction of correctly estimated impact set over the estimated impact set, and the recall R is the fraction of correctly estimated impact set over the correct impact set.

$$P = \frac{|M \bigcap A|}{|A|} \times 100\% \quad (1)$$

$$R = \frac{|M \bigcap A|}{|M|} \times 100\% \quad (2)$$

We carried out the evaluation for three granularities i.e. Statement (S), Method (M), and Type (T). For each permutation, we randomly chose 10 instances of source code entities across types, namespaces, and sub-projects. Thus, we carried out impact analysis for 90 different queries per subject system.

To create a set of benchmark results for comparison, we provided the source code of all four projects to three C# developers and asked them to analyze each subject system carefully for all the 90 queries. All the developers belong to Siemens CT DC AA and had on the average 6.3 years of programming experience. None of the chosen developers belong to projects P1 or P2. We illustrated the usage of Augur-Tool to all of them without providing details about how

TABLE II

RESULTS OF THE QUANTITATIVE EVALUATION: G_c REPRESENTS THE GRANULARITY (CHANGE) AND G_e REPRESENTS THE GRANULARITIES (EXPECTED). P_{avg}[S], P_{avg}[M], AND P_{avg}[T] SHOW THE AVERAGE PRECISION FOR THE GRANULARITY G_e AND S, M, AND T RESPECTIVELY. SIMILARLY, R_{avg}[S], R_{avg}[M], AND R_{avg}[T] SHOW THE AVERAGE RECALL FOR THE GRANULARITY G_e AND S, M, AND T RESPECTIVELY. VALUES IN SQUARE BRACKETS SHOW THE RESULTS OBTAINED FROM THE SIMPLE CIA METHOD.

	G_c	G_e	P_{avg}[S]	P_{avg}[M]	P_{avg}[T]	R_{avg}[S]	R_{avg}[M]	R_{avg}[T]
GitEx	S	S, M, T	0.50[0.50]	0.65[NA]	0.54[NA]	0.83[0.83]	0.83[NA]	0.75[NA]
	M	S, M, T	0.68 [NA]	0.58[0.55]	0.58 [NA]	0.85 [NA]	0.83[0.72]	0.89 [NA]
	T	S, M, T	0.63 [NA]	0.64 [NA]	0.63[0.61]	0.81 [NA]	0.81 [NA]	0.92[0.83]
VsVim	S	S, M, T	0.69[0.69]	0.44 [NA]	0.52 [NA]	0.72[0.72]	0.92 [NA]	0.79 [NA]
	M	S, M, T	0.56 [NA]	0.39[0.36]	0.58 [NA]	0.81 [NA]	0.83[0.72]	0.77 [NA]
	T	S, M, T	0.53 [NA]	0.53 [NA]	0.53[0.50]	0.76 [NA]	0.92 [NA]	0.78[0.67]
P1	S	S, M, T	0.50[0.50]	0.49 [NA]	0.58 [NA]	0.83[0.83]	0.73 [NA]	0.88 [NA]
	M	S, M, T	0.58 [NA]	0.59[0.50]	0.47 [NA]	1 [NA]	0.90[0.63]	0.75 [NA]
	T	S, M, T	0.58 [NA]	0.67 [NA]	0.55[0.55]	1 [NA]	1 [NA]	0.71[0.56]
P2	S	S, M, T	0.50[0.50]	0.55 [NA]	0.45 [NA]	1 [1]	1 [NA]	0.73 [NA]
	M	S, M, T	0.81 [NA]	0.55[0.50]	0.47 [NA]	0.92 [NA]	1 [0.83]	0.88 [NA]
	T	S, M, T	0.50 [NA]	0.45 [NA]	0.5 [0.42]	0.83 [NA]	0.83 [NA]	1 [0.75]

it internally works. They carried out a manual analysis and documented the potential impacted entities for all the queries for all subject systems independently. Next, we used Augur-Tool to analyze the change impact across granularities for each and every statement, method, and type for each project.

2) Evaluation Results: Table II shows the results of the quantitative evaluation. G_c represents the granularity of the source code entity that was intended to be changed and G_e represents the granularities of the expected potential source code entities. Each row summarizes the results of 30 different cases for 3 permutations. P_{avg}[S], P_{avg}[M], and P_{avg}[T] show the average precision for the granularity G_e and S, M, and T respectively. Similarly, R_{avg}[S], R_{avg}[M], and R_{avg}[T] show the average recall for the granularity G_e and S, M, and T respectively. To showcase the improvement in CIA that Augur brings, we compare it with a simple CIA method that supports only data and control dependencies and does not support variable granularities. Table II presents a comparison of accuracy between both the methods where square brackets show the results obtained from the simple CIA method.

Table II shows 55.5% and 81% average and maximum precision respectively exhibited by Augur for the randomly-generated test query sets. Further, the precision shown by Augur was considerably high compared to the simple CIA method in all four subject systems with a maximum improvement in precision of 18% with the associated recall improvement being 43% and an average improvement in precision of 6.1%. This indicates that Augur produced substantially more relevant than irrelevant results compared to the simple CIA approach.

Further, Augur reports 85.5% and 100% average and maximum recall respectively across all granularities. Augur shows considerably high recall compared to the simple CIA approach with a maximum improvement being 43% and an average improvement of 13.8%. In other words, Augur was better at returning most of the relevant results.

Although the Augur-Tool was able to produce the correct impact set most of the time, there were a few instances where the tool could not point out all the relevant impacted entities. We present such instances below:

- Augur includes a set of rules to detect environment dependency. However, due to the limitations of static analysis, some instances of such dependencies could not be inferred statically. For instance, one of the environment dependency rules detects a hidden dependency between two source code entities if one entity is reading from a file and another entity is writing to a file. However, if file names are not retrievable statically (for instance, when user is specifying the file names at run time) then the rule cannot detect the dependency between the entities.
- The current implementation of Augur provides rules for most common instances of environment dependency. We observed a few instances where an environment dependency (for instance, configuration dependency) could not be identified using the present rule-set.
- The current implementation of Augur-Tool does not account for a dependency when a type is used as a generic type parameter to a generic class.
- The tool is currently unable to report references of a source code entity that are coming from an XAML file in a C# project.

In the future, we plan to extend the current implementation of the Augur-Tool to address the limitations mentioned above.

IV. RELATED WORK

In this section, we present related work and compare Augur and Augur-Tool with existing CIA approaches and tools.

A. CIA through mining repositories and information retrieval

Many CIA approaches have been proposed based on mining software repositories and information retrieval techniques. These include methods proposed by Canfora et al. [4], Hattori et al. [5], Zimmermann et al. [6], Jashki et al. [7], Ceccarelli et al. [8], Canfora et al. [9], Ahsan et al. [10], Gethers et al. [11], and Kagdi et al. [12]. However, the impact analysis based on history may not be comprehensive and complete because the result naturally depends on the past activities and past dependencies. Such results cannot cover the dependencies introduced by the recent changes. Further, most of these approaches support impact analysis at a single granularity

level. Some of the approaches such as methods proposed by Hattori et al. [5] and Zimmermann et al. [6] cover more than one granularity; however, their coverage is limited to only three granularity levels. Finally, none of the above-mentioned approaches support inter-granular CIA queries. In contrast, Augur's comprehensive static code analysis supports six source code granularities and inter-granular CIA queries.

B. CIA through dependency analysis

Dependency analysis (static as well as dynamic) has been used by many existing CIA approaches to predict the impact set. Such approaches include ChAT [13], methods proposed by Breech et al. [14], Briand et al. [15], Badri et al. [16], Acharya et al. [17], Petrenko et al. [19], and Ajrnal et al. [18]. However, most of the existing approaches support only one source code granularity (such as class or method). A few approaches such as the method proposed by Petrenko et al. [19] extend the support for more than one granularity; however, to the best of our knowledge, a comprehensive CIA solution that supports all source code granularities during impact analysis does not exist. Further, none of the existing approaches mentioned above support inter-granular CIA queries. Finally, these dependency based methods for CIA rely only on techniques such as data and control dependency analysis which is often insufficient. They do not exploit hidden dependencies arising from semantic or environmental semantics as Augur does.

V. Conclusions

This paper introduced Augur, a more effective and comprehensive automated static code analysis approach for CIA. Augur performs CIA at all source code granularities and supports inter-granular CIA queries. It supports capturing semantic and environment dependencies in addition to data and control dependencies. The Change Impact Query Language provides a convenient and scalable mechanism to help developers execute multiple change impact queries in a single batch. Augur has been realized as Augur-Tool which is an extension for Visual Studio IDE. A quantitative evaluations has been conducted on the Augur-Tool to evaluate the accuracy of Augur. Evaluation results indicate that Augur reports source code entities that are impacted by a change with a higher-level of accuracy.

In the future, we plan to extend our current evaluation and augment Augur-Tool by covering more hidden dependencies.

Acknowledgment

The authors would like to thank Amit Patil and Kishan Kesavan, interns at Siemens CT DC AA who helped implement parts of the tool. We would also like to thank all the participants from Siemens who participated in the evaluation process of the tool and provided their valuable inputs.

References

[1] R. Glass, "Frequently forgotten fundamental facts about software engineering," *IEEE Software*, vol. 18, no. 3, pp. 112–111, May 2001.

[2] G. Samarthyam, G. Suryanarayana, T. Sharma, and S. Gupta, "Midas: A design quality assessment method for industrial software," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013, pp. 911–920.

[3] S. Bohner, "Impact analysis in the software change process: a year 2000 perspective," in *Proceedings of International Conference on Software Maintenance*, Nov 1996, pp. 42–51.

[4] G. Canfora and L. Cerulo, "Fine grained indexing of software repositories to support impact analysis," in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, 2006, pp. 105–111.

[5] L. Hattori, G. dos Santos Jr, F. Cardoso, and M. Sampaio, "Mining software repositories for software change impact analysis: A case study," in *Proceedings of the 23rd Brazilian Symposium on Databases*, 2008, pp. 210–223.

[6] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *Proceedings of the 26th International Conference on Software Engineering*, 2004, pp. 563–572.

[7] M.-A. Jashki, R. Zafarani, and E. Bagheri, "Towards a more efficient static software change impact analysis method," in *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2008, pp. 84–90.

[8] M. Ceccarelli, L. Cerulo, G. Canfora, and M. Di Penta, "An eclectic approach for change impact analysis," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, 2010, pp. 163–166.

[9] G. Canfora and L. Cerulo, "Impact analysis by mining software and change request repositories," in *11th IEEE International Symposium on Software Metrics*, Sept 2005, pp. 9–29.

[10] S. N. Ahsan and F. Wotawa, "Impact analysis of scrs using single and multi-label machine learning classification," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 2010, pp. 51:1–51:4.

[11] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk, "Integrated impact analysis for managing software changes," in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 430–440.

[12] H. Kagdi, M. Gethers, and D. Poshyvanyk, "Integrating conceptual and logical couplings for change impact analysis in software," *Empirical Software Engineering*, vol. 18, no. 5, pp. 933–969, 2012.

[13] M. Lee, "Change impact analysis of object-oriented software," Ph.D. dissertation, George Mason University, 1998.

[14] B. Breech, M. Tegtmeyer, and L. Pollock, "Integrating influence mechanisms into impact analysis for increased precision," in *Proceedings of the 22Nd IEEE International Conference on Software Maintenance*, 2006, pp. 55–65.

[15] L. Briand, J. Wust, and H. Lounis, "Using coupling measurement for impact analysis in object-oriented systems," in *Proceedings of IEEE International Conference on Software Maintenance*, 1999, pp. 475–482.

[16] L. Badri, M. Badri, and D. St-Yves, "Supporting predictive change impact analysis: a control call graph based technique," in *12th Asia-Pacific Software Engineering Conference*, Dec 2005, pp. 9 –18.

[17] M. Acharya and B. Robinson, "Practical andchange impact analysis based on static program slicing for industrial software systems," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 746–755.

[18] M. Ajrnal Chaumun, H. Kabaili, R. Keller, and F. Lustman, "A change impact model for changeability assessment in object-oriented software systems," in *Proceedings of the Third European Conference on Software Maintenance and Reengineering*, 1999, pp. 130–138.

[19] M. Petrenko and V. Rajlich, "Variable granularity for improving precision of impact analysis," in *IEEE 17th International Conference on Program Comprehension*, May 2009, pp. 10–19.

[20] Z. Yu and V. Rajlich, "Hidden dependencies in program comprehension and change propagation," in *Proceedings of 9th International Workshop on Program Comprehension*, 2001, pp. 293–299.

[21] T. Sharma, "Identifying extract-method refactoring candidates automatically," in *Proceedings of the Fifth Workshop on Refactoring Tools*, 2012, pp. 50–53.

[22] "Microsoft Visual Studio," http://www.visualstudio.com/, [Online; accessed on 20-June-2016].

[23] "NRefactory," https://github.com/icsharpcode/NRefactory, [Online; accessed on 20-June-2016].

[24] "GitExtensions - A shell extension as well as Visual Studio plug-in for Git repository," https://github.com/gitextensions/gitextensions, [Online; last accessed on 20-June-2016].

[25] "VsVim - Vim emulator plug-in for Visual Studio," https://github.com/jaredpar/VsVim/, [Online; last accessed on 20-June-2016].