IJISET - International Journal of Innovative Science, Engineering & Technology, Vol. 1 Issue 4, June 2014.
www.ijiset.com
ISSN 2348 - 7968

A Review on Specification Mining Architecture

Sangeetha K [1], Prof. Pankaj Dalal[2]

[1] M Tech Scholar (Software Engineering), Department of Computer Engineering, Shrinathji Institute of Technology & Engineering, Nathdwara-313301, India

[2] Associate Professor,), Department of Computer Engineering, Shrinathji Institute of Technology & Engineering, Nathdwara-313301, India

Abstract

To aid program testing efforts and help program verification tools to find bugs and ensure correctness of systems the mined specifications can be used. Mining of specification process starts with a program under analysis and/or a set of test cases. By applying static or dynamic analysis method traces should be produced from the program. Techniques employing dynamic analysis require the running of the test cases to produce a set of traces which is later analyzed. Program traces are often long and involve several phases. Semantic clustering can be used to split the traces vertically into phases based on comments or annotations on source code. This paper shows a review on the architecture of the automaton based specification mining process.

Keywords: *Specification mining, filtering, clustering, LSI*

1. Introduction

Software bugs are prevalent. Bugs not only make it more expensive in developing software systems due to the high cost involved in debugging, but also may cause various security vulnerabilities. The absence of specifications has made it harder to locate bugs. Many existing bug finding tools, e.g., model checking, require the availability of specifications in order to locate bugs which are defined as anomalies or violations of these specifications.

To address the above challenges (i.e., to improve program understanding and to find bugs), specification mining has been proposed. It is a program analysis method to automatically infer the specification of a program based on examples of correct usage. Usage refers to the manner in which the program or its exposed methods are invoked. For example, correct usage of resources such as a file or new connection follows acceptable invocation sequence: acquisition, access and then release. Similarly to use individual methods correctly, the parameters passed to it should meet the necessary preconditions. These are the implicit rules, followed by most programs but not explicitly stated, that mining techniques attempt to uncover.

The mining of various specification formats such as automata and temporal rules has been developed. In general, specification mining techniques employ data mining or machine learning on execution traces to generate models that are useful in program verification. These techniques work under the assumption that by observing sufficient executions of a good software implementation, inferences regarding the specification (or expected behavior) of the software can be made.

Specification mining starts with a program under analysis and/or a set of test cases. There have been both dynamic and static approaches for specification mining. Broadly, dynamic specification mining techniques rely on actual executions of programs. In contrast, static approaches look to extract the specification by reasoning on the control flow of a subject program or of other 'client' programs that invoke the subject. Static specification mining can be performed if program source code is available. However, to obtain precise specifications, expensive analysis may have to be performed to eliminate infeasible paths. This obstacle is more overwhelming in the distributed case, where feasible scenarios (the number of processes and how they will interact) have to infer based on the static view provided by the program source code executed by each process.

2. Background

The term specification mining is first coined by Ammons et al.in [1]. According to him, Specification mining is a process of inferring models or properties that hold for a system. It discovers some of the temporal and data-dependence relationships that a program follows when it interacts with an application programming interface (API) or abstract data type (ADT).

Alur et al. propose an approach to infer a finite state machine from Application Programming Interface (API) code [3]. Mariani and Pezz´e[4] propose a new grammar inference engine specially suited for program executions named k-behavior. Acharya et al [5] propose a static analysis approach to mine finite state machines from systems. Their tool leverages a model checker to statically generate traces of a system. Dallmeier et al [6] propose a hybrid static and dynamic analysis approach to infer a finite state machine. First, a set of methods termed as mutators (i.e., those that change a system state) and another set termed as inspectors (i.e., those that read/inspect a system state) are determined statically Acharya et al. extend their previous work [5] in [8]. Static traces are first extracted

ISSN 2348 - 7968

from program code similar to their previous work. Relevant segments of traces are then recovered. These segments of traces fed to a frequent partial order miner. The resultant partial orders are then composed to generate a specification in the form of a finite state machine.

Mariani et al. extend their previous work [4] in [9]. They propose a technique to generate prioritized regression test cases for the integration of Commercial-off-the-Shelf (COTS) components. A model in the form of a finite state machine and a set of boolean expressions on a set of variables, operators, and values is learned based on an older version of a system. Lorenzoli et al. mine extended finite state machines (EFSMs) in [10]. An extended finite state machine enriches a standard finite state machine with value-based invariants. Mariani and Pastore mine finite state machines to identify failure causes from system logs[11]. System logs are first collected. Several pre-processing modules to detect for events in logs and to transform data to an appropriate format that abstracts away concrete values are first employed.

Lo and Khoo extend the work by Ammons et al.[1] by proposing a metric of precision and recall in evaluating the quality of a specification mining engine producing finite state machines[7]. They also introduce trace clustering and trace filtering to reduce the effect of bad traces and inaccuracies during the inference of mined specifications. They developed a mining architecture called SMArTIC which is Specification Mining Architecture with Trace fIltering and Clustering and is a API specification mining architecture used to improve the accuracy, robustness and scalability of specification miners.

3. Mining Process

Mining process begins with traces of a program's run-time interaction with an API or ADT. According to Ammons et al. [1] specification mining system is composed of four parts: tracer, flow dependence annotator, scenario extractor, and automaton learner (Figure1).

The tracer instruments programs so that they trace and record their interactions with an API or ADT, as well as compute their usual results. The tracers produce traces in a standard form, so that the rest of the process is independent of the tracing technology.

Flow dependence annotation is the first step in refining the traces into interaction scenarios, which can be fed to the learner. It connects an interaction that produces a value with the interactions that consume the value. Next, the scenario extractor uses these dependences to extract interaction scenarios—small sets of dependent interactions—and puts the scenarios into a standard, abstract form.

The automaton learner is composed of two parts: an off-the-shelf probabilistic finite state automaton (PFSA) learner and a post processor called the corer.

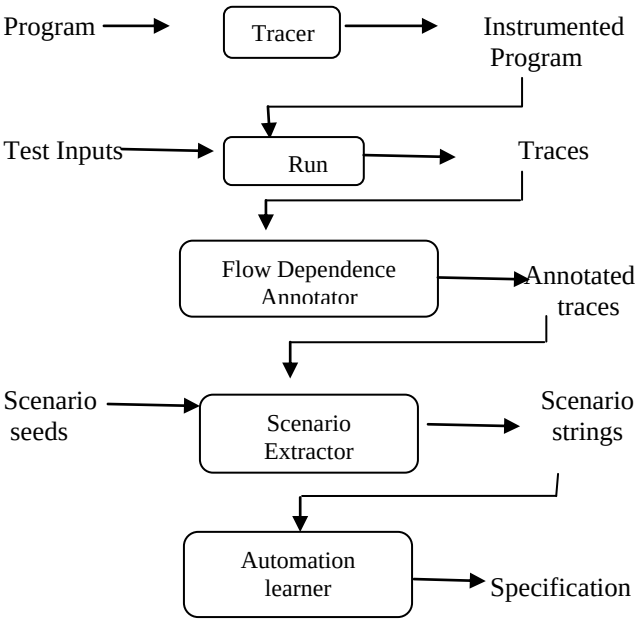

Fig 1: Specification Mining System

4. Mining Architecture

Lo and Khoo proposed a novel architecture for automaton based specification mining [7]. It explores the art and science behind the construction of such a miner. It achieves specification mining through pipelining of four functional components: Error-trace filtering, clustering, learning, and automaton merging.

This architecture known as SMArTIC is Specification Mining Architecture with Trace fIltering and Clustering which is a API specification mining architecture used to improve the accuracy, robustness and scalability of specification miners.

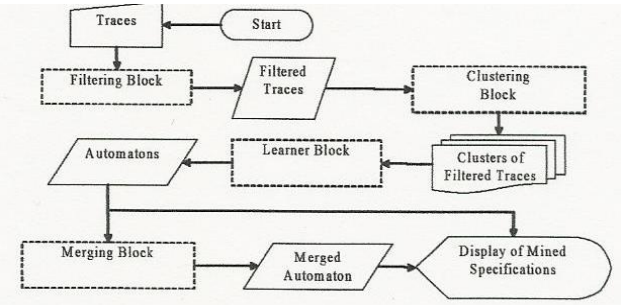

Fig 2: SMArTIC Structure

The overall structure of SMArTIC is as shown in Figure 2. It comprises 4 major blocks, namely filtering, clustering, learning and merging blocks. Each block is in turn composed of several major elements. The filtering block filters erroneous traces to address the robustness issue. The clustering block divides traces into groups of "similar" traces to address scalability issue. The learning block

ISSN 2348 - 7968

generates specifications in the form of automata. The merging block merges the automatons generated from each cluster into a unified one.

4.1 Filtering Block

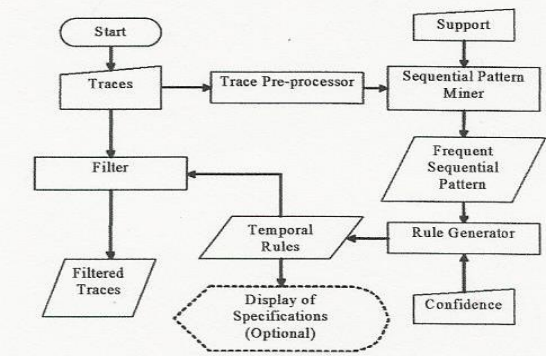

Fig 3: Filtering Block

The filtering block aims to filter out erroneous traces based on common behavior found in a multi-set of program traces. Since a trace is a temporal or sequential ordering of events, representing common behavior by "statistically significant" temporal rules will be appropriate. Certainly, temporal rules based on full set of temporal logics will be a good candidate, but it is desirable to have a more light-weight solution.

Implementation-wise, the structure of the filtering block is as shown in Figure 3.

4.2 Clustering Block

Input traces might be mixed up from several unrelated scenarios. Grouping unrelated traces together for a learner to learn might multiply the effect of inaccuracies in learning a scenario. Such inaccuracies can be further permeated into other scenarios through generalization. The clustering block converts a set of traces into groups of related traces. Clustering is meant to localize inaccuracies in learning one sub-specification and prevent the inaccuracies from being permeated to other sub-specifications. Furthermore, by grouping related traces together, better generalization can be achieved when learning from each cluster. Two major issues pertaining to clustering are: the choice of clustering algorithm and an appropriate similarity metric; i.e., measurement of similarity between two traces. The performance of the clustering algorithm is affected by appropriate similarity/distance metric. The general structure of the clustering block is as shown in Figure 4.

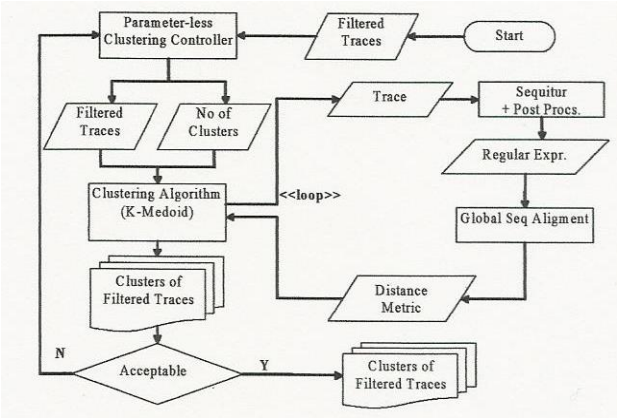

Fig 4: Clustering Block Structure

4.2.1 Clustering Algorithm

A classical off-the-shelf clustering algorithm, namely the k-medoid algorithm is used in SMArTIC. The k-medoid algorithm works by computing the distance between pairs of data items based on a similarity metric; this corresponds to computing the distance between pairs of traces. It then groups the traces with small distances apart into the same cluster. The k in k-medoid is the number of clusters to be created. In SMArTIC, the Turn∗ algorithm presented by Foss is adapted into the k-medoid algorithm. The Turn∗ algorithm can automatically determine the number of clusters to be created by considering the similarities within each cluster and differences among clusters. This algorithm will repetitively increase the number of clusters. For each repetition, it will divide datasets into clusters and evaluate a measure of similarities within each cluster and differences among different clusters. The algorithm will terminate once a local maximum is reached.

Similarity metric for the program traces is calculated from the regular expression of the program traces. The regular expression representation will be obtained by converting a trace to its hierarchical grammar representation using Sequitur. The output of sequitur will be post-processed to construct the regular expression representation and then be fed in as input to global sequence alignment. With these a method to find a reasonable distance metric for the similarity of program traces will be obtained. Generated sequitur grammar of a trace will be processed through several passes of reduction to produce a regular expression representation.

4.3 Learning Block

Although temporal rules have also been used to capture certain information of a program specification, automata have been commonly used in capturing specifications, especially protocol specifications. The purpose of this learning block is to learn automatons from clusters of filtered traces. This block is actually a placeholder in the

architecture. Different PFSA specification miners can be placed into this block, as long as they meet the input-output specification of a learner. Once a learner is plugged in, it will be used to mine the traces obtained from each cluster. At the end, the learner produces one mined automaton for each cluster. Sk-strings learner is used by Ammons [1] to mine the specification of the X11 windowing library. It is an extension of the k-tails heuristic algorithm of Biermann and Feldman for learning stochastic automata. In k-tails, two nodes in a constructed automaton are checked for equivalence by looking at subsequent k-length strings that can be generated from them. Different from k-tails, in sk-strings, subsequent strings need not necessarily end at an end node, except for strings of length less than k. Furthermore, only the top s% of the most probable strings that can be generated from both nodes is considered. Implementation-wise, the sk-strings learner first builds a canonical "machine" similar to a prefix-tree acceptor from the traces. The nodes in this canonical machine are later merged if they are indistinguishable with respect to the top s% of the most probable strings of length at most k that can be generated starting from them.

4.4 Merging Block

The merging process aims to merge multiple PFSAs produced by the learner into one such that there is no loss in precision, recall and likelihood before and after the merge. Equivalently, the merged PFSA accepts exactly the same set of sentences as the combined set (i.e., union) of sentences accepted by the multiple PFSAs. The primary purpose of the merging process is to reduce the number of states residing in the output PFSA by collapsing those transitions behaving "equivalently" in two or more input PFSAs, thus improving scalability.

Merging of two PFSAs involves identification of equivalent transitions between the two PFSAs. The output PFSA contains all transitions available in the input PFSAs, with each set of equivalent transitions represented by a single transition.

Merger block will merge PFSAs produced by learner block to a unified one. The merging is performed in the "safe" way where no further generalization is performed during the merging process. Hence the set of language accepted by the final PFSA will be the union of the set of languages accepted by each of the PFSAs. The merging process is performed iteratively by merging 2 PFSAs at a time. Let's call them X and Y for ease of reference.

There are 6 steps in automaton merge algorithm. They are

1. Handling of exception cases,
2. Creation of unifiable list,
3. Creation of mergable list,
4. Creation of equivalence class,
5. Creation of structural merge
6. Creation of full merge.

5. Clustering Enhancement

Lo and Khoo has proposed an architecture which groups the similar traces into cluster (horizontal split). Program traces are often long and involve several phases. Now semantic clustering can be used to split the traces vertically into phases based on comments or annotations on source code. By merging the idea of SMArTIC and semantic clustering, it may be possible to split the traces not only horizontally (splitting traces into separate clusters) but also vertically (splitting traces into phases). This will leverage the benefit of divide and conquer approach in the specification mining process. i.e, in the mining process after filtering the erroneous traces from the set of traces, it should be fed to the clustering block.

5.1 Semantic Clustering

This clustering process uses an information retrieval technique called Latent Semantic Indexing [15, 17]. The semantics of the source code i.e., the names of identifiers, comments etc. can be analyzed by using this LSI [15].

LSI is an indexing and retrieval method that uses a mathematical technique called singular value decomposition (SVD) to identify patterns in the relationships between the terms and concepts contained in an unstructured collection of text. LSI is based on the principle that words that are used in the same contexts tend to have similar meanings. A key feature of LSI is its ability to extract the conceptual content of a body of text by establishing associations between those terms that occur in similar contexts.

We can use LSI to retrieve the semantic similarity between different entities (i.e., whole systems, classes and methods) and then we can cluster these entities according to their similarity. i.e., we can characterize a system by clustering its classes or a class by clustering its methods.

LSI takes as input a collection of text documents and yields as output an index with similarities between these documents. LSI will automatically label the clusters with their most relevant terms.

It is called Latent Semantic Indexing because of its ability to correlate semantically related terms that are latent in a collection of text; it was first applied to text at Bell Laboratories in the late 1980s. The method, also called latent semantic analysis (LSA), uncovers the underlying latent semantic structure in the usage of words in a body of text and how it can be used to extract the meaning of the text in response to user queries, commonly referred to as concept searches. Queries, or concept searches, against a set of documents that have undergone LSI will return results that are conceptually similar in meaning to the search

criteria even if the results don't share a specific word or words with the search criteria.

By merging the idea of SMArTIC and semantic clustering, it may be possible to split the traces not only horizontally (splitting traces into separate clusters) but also vertically (splitting traces into phases). This will leverage the benefit of divide and conquer approach in the specification mining process

6. Conclusion and Future Work

This paper presented a review on the architecture of the specification mining. Specification mining is a process of inferring models or properties that hold for a system. It starts with a program under analysis and/or a set of test cases and also involves the running of the test cases to produce a set of traces which is later analyzed. This topic has been ongoing for over a decade and more than 50 papers have been published on this topic. These papers can be classified into studies on the extraction of finite state machines, works that mine for value-based invariants, works that mine rules and patterns and studies extracting sequence diagram like specifications.

Specification mining process – originated in the field of software engineering and programming language – can be improved by the synergy of various computer science domains: data mining, software engineering, programming language, learning theory, automata theory, etc., resulting in: new and more objective evaluation frameworks, more accurate mining results, more compact mining results, scalable and manageable mining processes, automation of manual processes, novel applications and even has impact on the original domains where several techniques used are originated.

D.Lo and S. C. Khoo has proposed a novel architecture called SMArTIC [7] which addresses the research issues like accuracy, robustness and scalability. They proposed a metric of precision and recall in evaluating the quality of a specification mining engine producing finite state machines. They also introduce trace clustering and trace filtering to reduce the effect of bad traces and inaccuracies during the inference of mined specifications.

As future work we can split the traces both into clusters as well as phases. This will localize the inaccuracies in learning one sub specification and prevent the inaccuracies from being permeated to other sub specifications. Also by grouping related traces together, better generalization can be achieved when learning from each cluster.

The success of this process depends upon the choice of the algorithms. For splitting the traces into clusters or splitting the traces horizontally, we can use k- medoid algorithm as used in SMArTIC [7]. Then the splitted traces will be given to retrieve the semantic similarity between different traces and we cluster these traces according to their similarity into different phases. i.e., we can extend the clustering process of specification mining, by using the concept of Latent Semantic Indexing for splitting the traces into phases. This will leverage the benefit of divide and conquer approach in the specification mining process.

REFERENCES

[1] G. Ammons, R. Bodik, and J. R. Larus. *Mining specification*. In Proc. of Principles of Programming Languages, 2002

[2] J.E. Cook and A.L. Wolf. *Automating process discovery through event data analysis*. In Proceedings of ACM/IEEE International Conference on Software Engineering, pages 73–82, 1995.

[3] R. Alur, P. Cerny, G. Gupta, and P. Madhusudan. *Synthesis of interface specifications for java classes.* In Proceedings of ACM Symposium on Principles of Programming Languages, pages 98–109, 2005.

[4] L. Mariani and M. Pezz`e. *Behavior capture and test: automated analysis for component integration*. In Proceedings of IEEE International Conference on Engineering of Complex Computer Systems, pages 292–301,2005.

[5] M. Acharya, T. Xie, and J. Xu. *Mining interface specifications for generating checkable robustness properties*. In Proceedings of International Symposium on Software Reliability Engineering, pages 311–320, 2006.

[6] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. *Mining object behavior with ADABU*. In Proceedings of International Workshop on Dynamic Analysis, pages 17–24, 2006.

[7] D. Lo and S-C. Khoo. *SMArTIC: Specification mining architecture with trace filtering and clustering*. In SoC-NUS tech. report, TRA 8/06, 2006.

[8] M. Acharya, T. Xie, J. Pei, and J. Xu. *Mining API patterns as partial orders from source code: from usage scenarios to specifications*. In Proceedings of Joint Meeting of the European Software Engineering Conference and the ACM International Symposium on Foundations of Software Engineering, pages 25–34, 2007.

[9] L. Mariani, S. Papagiannakis, and M. Pezz`e. *Compatibility and regression testing of COTS-component-based software*. In Proceedings of ACM/IEEE International Conference on Software Engineering, pages 85–95, 2007.

[10] D. Lorenzoli, L. Mariani, and M. Pezz`e. *Automatic Generation of Software Behavioral Models.* In Proceedings of ACM/IEEE International Conference on Software Engineering, pages 501–510, 2008.

[11] L. Mariani and F. Pastore. *Automated identification of failure causes in system logs.* In Proceedings of International Symposium on Software Reliability Engineering, pages 117–126, 2008.

[12] W. Damm and D. Harel. *LSCs: breathing life into message sequence charts*. Journal on Formal Methods in System Design, 19(1):45–80, 2001.

[13] A. Kuhn, S. Ducasse, and T. Girba. *Enriching reverse engineering with semantic clustering.* In Proc. of Work. Conf. on Reverse Engineering, 2005.

[14] Douglas E.Comer and David L.Stevens. *Internetworking with TCP/IP.Client-server Programming and Applications*, BSD Socket Version. Prentice-Hall,Englewood Cliffs,NJ 07632, USA,1993.

[15] A. Kuhn, S. Ducasse, and T. Girba. *Enriching reverse engineering with semantic clustering*. In Proc. of Work. Conf. on Reverse Engineering, 2005.

[16] Mining Software Specifications: Methodologies and Applications **By:** David Lo; Siau-Cheng Khoo; Jiawei Han; Chao Liu0

[17] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Fur- nas, and R. A. Harshman. Indexing by latent semantic analysis. Journal of the American Society of Information Science, 41(6):391–407, 1990.