A Systematic Judgement to Automated Programming Contracts Generation

S.V.Gayetri Devi, C.Nalini

ABSTRACT--- Contracts provide a pre-emptive approach in identifying programming errors at run-time using assertions or by formal Static analysis tool or Manual source code reviews. They describe the expected software behavior. Contracts written by developers have a greater error detection ability than the generic ones that are created automatically but may involve strenuous efforts for larger sized source codes. The intent of this paper is a concise study of prevalent approaches in the generation of contracts and to put forward an approach to derive programming rules for real-time concurrent Java source code automatically with reduced efforts. The proposed method extracts the scalar variables and computed constants from Static program code analysis, then identifies various dependencies dynamically and generates the declarative contracts automatically by Decision tree modeling of computed dependencies. These rules can then be utilized for software Verification.

Index Terms — Contracts, Software Analysis, dependencies, Decision trees, Optimization

I. INTRODUCTION

The concept of contracts was at first presented by Meyer [1] as an order of responsibilities with determined pre and post conditions. If this order is performed without complying with these conditions, the contract is said to be violated. Contracts in the form of regular expressions can be used to identify protocols for acquiring objects in sequential [2] as well as concurrent developments [3, 4, 5]. Contracts thus specify what aspects, the software should fulfil during the execution of program. They consist of preconditions, post conditions and invariants.

The precondition of a subprogram, expresses the conditions to be met by the caller functions before any of its executions. If the precondition is not satisfied, the segment of the code may not carry out its intended cause as the function does not hold accountable for the outcome which may be undefined operations and terminations. Guards or assertions can be used to test the preconditions within the code itself because there is no mechanism inside the function to confirm if the precondition is satisfied. The function only believes that a verification has already been performed before the function is invoked. In object oriented context, preconditions for any method, define any restriction on the state of the object required for successful execution. A precondition may be with respect to a subprogram or an operation in formal specification. Preconditions on public routines are imposed by explicit checks inside methods causing stated exceptions. An assertion is unsuitable for such preconditions, as the enclosing method assures that it will impose the parameter checks, irrespective of whether the assertions are enabled or not. For preconditions on nonpublic methods with the programmer on view that the precondition must be satisfied, then the assertion is apt.

A post condition states what a program routine is to provide when a right function invocation is done. A function invocation is right when the preconditions satisfy the requirement when a call is done on it. Here, the function is accountable for the outcome and the client routine can expect that the post condition is held valid. The post condition of a function indicates that the conditions to be exhibited by the function after it is executed.

Invariant relates to a logical condition that is said be held true always during a particular phase of program execution. It must be fulfilled in every visible state. It serves to strengthen the preconditions as well as postconditions of all program operations. In OO context, an object invariant comprises of properties that stay rigid irrespective of the object's state.

Contracts in the form of preconditions (to detect and separate out unacceptable inputs) and postconditions (that function as test oracles) enable automated testing. They also express about what a program operation expects from its arguments and what outcome it produces. Most of the programming languages have this idea specified as Comments. Such semantics are represented in standard formats and the intent can be verified : (i) through Manual code reviews at instants when Run time assertion checking is not supported (ii) Formal static analysis by the use of various tools to verify that precondition is satisfied at each invocation and postcondition is valid based on the operation implementation and (iii) Run time using assertion checking to confirm if the precondition is complied with at method entry point and postcondition is held on return.

Two software components are said to have dependency between if a change to one may have an effect on another requiring changes.

Dependency analysis is the method of identifying the dependences of a software program [20]. Callo Arias et al. [21] categorizes dependencies to be structural, behavioral, and traceability. Structural dependencies are derived through static analysis of source code. In Object oriented context, structural dependencies include Import, Declaration, Call, Access, Inheritance, and Annotation. Behavioral dependency analysis appraises the extent to which the functionality of a module is dependent on other modules.

Revised Manuscript Received on July 10,2019.

S.V.Gayetri Devi, Research Scholar Department of Computer Science and Engineering Bharath Institute of Higher Education and Research Chennai, T.N, India. (gayetri.venkhatraman@gmail.com)

C.Nalini, Professor Department of Computer Science and Engineering Bharath Institute of Higher Education and Research Chennai, T.N, India.

Since testing of all modules and operations are not feasible for very large software systems, the modules on which other modules are the most dependent must be tested primarily. Dependency analysis based on code and execution traces are used in testing, debugging and optimization of programs. Behavioral dependency can also be derived from the design model of the software.

To visualize and learn the dependencies, Decision trees can be used. A decision tree explains a Boolean or discrete function. A test is marked for every non leaf node of the tree. Either 1 or 0 is assigned to the leaf. The Boolean function is then evaluated by beginning at the root. If Test passes, left child of non-leaf node is visited and if test fails, right child is visited. The trail from root to leaf signifies the dependencies as assertions conditions [24]. Thus the decision tree can be transformed as Decision rules commonly of the form:

If condition#1 and condition#2 and condition#3 then decision outcome

These decision rules correspond to the contracts derived for software verification.

II. LITERATURE REVIEW OF CONTRACTS GENERATION TECHNIQUES

This segment presents a review of the prevalent approaches to derivation of Contracts.

Zhang et al., (2017) proposed a process to generate assertions during design verification of Integrated circuits. RTL ATPG algorithm is used for generation through Static approach. The paper targeted on complete coverage of the design specification

Shobha et al., (2010) presented GOLDMINE tool for creating assertions in an automated way. The tool encompasses Data mining and static analysis of the Register Transfer Level design.

Dias et al., (2015) describes that the contract of a program module is stated by the developers as an order of invocations of public methods that must be executed atomically to evade atomicity violations bugs in the module’s client program code. Based on concept of Programming by contract methodology, the ordering of methods to be executed atomically is the responsibility of the programmer.

E.Silva et al., (2015) showed that annotations manually inserted in the source code by the programmers served to express the contracts between the functions as per the software specifications.

Signoles et al., (2017) E-ACSL run time verification tool within Frama-C framework detailed the contracts in terms of safety and security properties, good order of method calls, any undefined behaviors if any, null pointer deference checks, valid memory accesses. These aspects can be indicated as annotations in source code written using the ECSL specification language**.**

Alshanqiti (2015) proposed a dynamic methodology to reverse engineer visual contracts from single threaded Java programs through the execution traces of Java program operations. The resultant contracts provide precise accounts of the perceived object transformations, their properties and preconditions in terms of object configurations, argument and attribute values, and allow generalization by multi objects.

Carr et al., (2017) proposed a CodeContractBot CCBot tool that employs a Static analysis tool to assure the insertions of contracts automatically inside code after analysis for bugs. In the perspective of CCBot, a false positive means a correct valid execution of the original program is present and it is shown to be infringing the contract. The static analyzer tool collects a restricted specification of the program’s behavior via existing contracts and implicit rules. If the static analyzer finds a contract which does not hold valid, that identifies an error in the program. The tool gives a warning when it cannot ascertain that a contract is satisfied.

TABLE I
COMPARATIVE STUDY ON DIFFERENT VERIFICATION MODELS IN CONCURRENT SOFTWARE

Paper & Author	Contract Specification	Method of Contracts creation	Application/ Merits	Limitations
Automatic Assertion Generation for Simulation, Formal Verification and Emulation Zhang et al., (2017)	Assertions are based on Register Transfer Level design representation. Only True assertions covering the functional specification are created.	To create assertions, symbolic expression is arrived at covering the functional implementation. Automatic Test Pattern Generation algorithm is employed to determine combinations of input to validate output.	No false assertions are produced. Time utilized for Verification is minimized.	Only TRUE assertions for the functional specifications of the design are generated.

GoldMine: Automatic assertion generation using data mining and static analysis Shobha Vasudevan et al., (2010)	Focuses on automatic generation of assertions by analysing the code statically and using decision trees	Traces of simulating the program module or design are first generated. Details about the design pertaining to domain are extracted. The details are then assimilated into mining algorithms to create assertions.	Assertions relating to complex design relationships are captured	Assertions are extracted from design. Tool is unable to detect Bugs pertaining to Design not conforming to specifications
Preventing Atomicity Violations with Contracts Dias et.al, (2015).	1. Define a contract as finite number of method call sequences from multithreaded java programs. 2. For methods considered: add(obj), contains(obj), indexOf(obj), get(idx), set(idx, obj), remove(idx), and size(), an example contract is 1. contains indexOf 2. indexOf (remove | set | get) 3. size (remove | set | get) 4. add indexOf.	Automated contracts generation: 1. Identify the sequence of calls made atomically in at least two places of source code. 2. Treat these sequences as contracts 3. Manually refine to exclude inappropriate contracts.	Applicable for prevalent situations of atomic violations in Java programs.	Class scope mode employed and contracts defined for the control flow within a class. Contracts pertaining to calls to other classes are not considered.
Formal Verification With Frama-C: A Case Study in the Space Software Domain e silva et.al, (2015).	Contracts between functions are realized as annotations placed in the C program code.	Manual code annotations for : 1. Safety specification writing for memory, integer, floating point and termination checks. 2. Functional specification related to postconditions of a function called on satisfying its described preconditions	Once the annotations are written manually, the Jessie plug-in enables Deductive verification of C programs inside Frama-C framework.	Very tedious to generate annotations manually for larger program size and complicated code.
E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs Signoles et.al., (2017)	Annotate the C source code with formal specifications of safety and security properties using E-ACSL language	Frama-C has plug-ins to produce E-ACSL annotations from high level specifications automatically and manual annotation is not required in most usages.	The E-ACSL specification language enables contract writing of various issues like proper ordering of function calls, data flow leakage, integer overflows and memory faults. Frama-C can generate most of these automatically. E.g. RTE plug-in to create annotations to check run time undefined behaviours.	Has no provision for identifying concurrency bugs like deadlocks or data races

Extracting Visual Contracts from Java Programs Alshanqiti et.al., (2015)	1. Analyse and learn Visual contracts (VCs) from single threaded Java source codes by tracing the program's execution. 2. The derived contract depicted the object transformations, preconditions, effects and established specifications. 3. Each contract instance comprises of a pair of object graphs signifying the state before and after the operation	1. Extract a contract instance for each call of program operation depicting the perceived behaviour by using AspectJ for to instrument the code and obtain traces. 2. Derive at concise contracts across sets of objects with different multiplicities by introducing multi objects.	Implemented as prototype tool with a Tracer to derive contract instances, a Generaliser for minimal, maximal and Mo rules, a Visualizer for contracts display and analysis.	1. Correctness may not hold for condition checking of state of an object outside the scope of selected implementation classes for contract extraction. 2. Completeness fails for behaviours not detected and considered in the model. 3. Does not support tracing and Contract extraction for concurrent java programs.
Automatic Contract Insertion with CCBot Carr e.al., (2017)	Instruments C# source code with Contracts as method pre and post conditions and object invariants using CodeContracts way after Static Analysis	1. Implements source-to-source translation of source code and automatically inserts the contracts as annotations instead of presenting as warnings 2. The instrumented contracts identify and avoid null-pointer dereferencing, buffer and integer overflows, and floating point precision discrepancies 3. The contracts can be added incrementally by considering the implicit contracts	1. Implemented as CodeContractsBot (CCBot) tool, the developer can see the contracts in appropriate perspective and can modify or test the code in automated way. Thus the tool acts as organized bug identification and source code quality improvement tool.	1. Need for a provision to integrate any generic Static analysis tool with CCBot. 2.Need for generalization of CCBot to other languages like Java with JML

III. REVIEW FINDINGS

Despite the several advantages of Contracts, only a smaller portion of the source code is contracted. Many programming languages have no built-in provision for contracts with developers having to use assertions manually to include conditions for run-time checks. But languages based on Design by Contract (DbC) like Java Modeling Language (JML) [6], Eiffel [7], Spec# [8] etc. and their linked IDEs provide for the language's capability to furnish methods with preconditions and postconditions semantics, classes with invariants, well expressed rules of Inheritance and also gives compiler - choices to enable or disable contracts checks at run time. To address the complexity of having contract included programs in languages with no DbC support, an automated way of generating contracts is desired. Also manually generated contracts are not scalable for complex and larger sized programs.

IV. IV. PROPOSED APPROACH FOR AUTOMATED CONTRACTS GENERATION

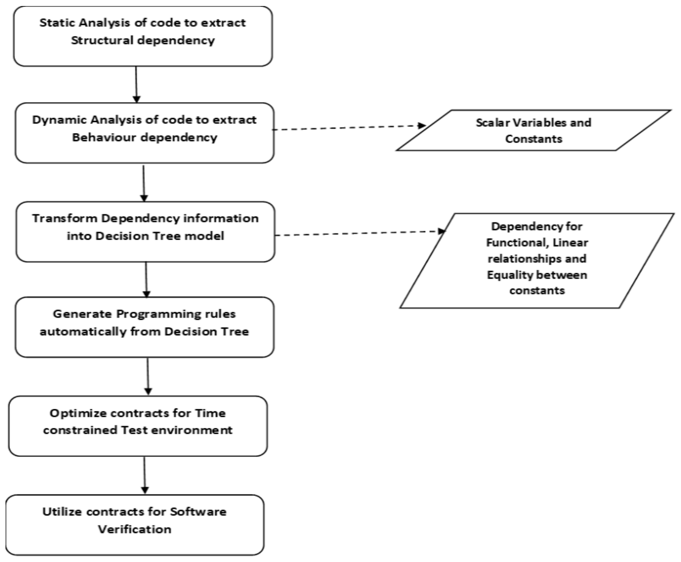
Static Analysis of code to extract Structural dependency
Dynamic Analysis of code to extract Behaviour dependency
Scalar Variables and Constants
Transform Dependency information into Decision Tree model
Dependency for Functional, Linear relationships and Equality between constants
Generate Programming rules automatically from Decision Tree
Optimize contracts for Time constrained Test environment
Utilize contracts for Software Verification

Figure 1. The proposed Programming contracts generation

The proposed method provides an automated approach to generate contracts. In our work, we propose AutoAssertMine, a methodology to automatically generate assertions using Static and Dynamic analysis. The test program code is statically analyzed to obtain semantic information representing the structural dependencies at statement level and module level like scalar variables, computed constants, classes, methods and their operations. Every information of the program code is revealed at run time. Hence behavioral dependencies namely Direct and Indirect, Internal, External, Indirect internal and Indirect external are determined through Dynamic code analysis. The frequency of methods that are invoked by other methods is also found. The frequency computation is essential to derive rules for method calls to detect any potential atomicity bug if some of the functions cannot be called by other methods as per the specifications. The extracted-Behavior dependency pertaining to three aspects - functional dependency, linear relationships, and equality between constants, the structural dependencies and the computed frequency are then modeled as Decision tree by storing the dependency information as assertion conditions in Decision tree. The conditions in the Decision tree are then directly transformed into a set of programming rules or contracts based on software specifications. Assertion generation is performed in lightweight manner. Numerous feedback circuits from different phases of the application source code are monitored by Metrics. Finally, a set of candidate statistical assertions are attained by ranking the AutoAssertMine assertions by means of metrics to assess support and confidence.

Mapper tree based supervised knowledge learning algorithm is utilized in AutoAssertMine. A mapping tree is comprised of internal nodes and terminal leaves. Each root node holds a decision function and the discrete results are represented by the leaf nodes. This ordered mapping process splits the input data space into local sections iteratively until a leaf node is reached in the entire binary tree mapping.

Each leaf node in the mapper tree turn out to be a candidate assertion.

The hit rate of a run in AutoAssertMine is the ratio of true assertions to candidate assertions. Once the assertions have been created through AutoAssertMine, their assessment is very vital. The machine produced outcomes are compared to those generated by humans and feedback is provided to AutoAssertMine to enhance its results accordingly. The set of contracts are then filtered and optimized to prioritize feasible contracts for time constrained testing efficiency and can be utilized for Software Verification through Abstract Interpretation and Deductive Verification.

Input Software Source used during contracts generation is:

https://github.com/nardevar/Banking

V. PROPOSED ALGORITHM FOR AUTOMATED CONTRACTS GENERATION

A. Code Analyzer

```
Begin
    For each java program do
    begin
            Initialize CCNT ←1,
            Repeat till readline ≠ NULL do
            CCNT ← CCNT+1;
    End
    begin // method counter
    Initialize MCNT ←1
    if readline ends with ")" then
            MCNT ← MCNT+1;
    end
    begin // abstract class counter
    ACCNT← 1
    Repeat till readline ≠ NULL do
    if readline has "class" && "abstract" then
            ACCNT← ACCNT +1
    End
    begin // method counter
    AMCNT←1;
    Repeat till readline ≠ NULL do
    if readline endswith ")" && readline has
    "abstract" then
            AMCNT← AMCNT +1
    End
    begin // interface counter
    ICNT ←1;
    Repeat till readline ≠ NULL do
    if readline contains "interface" then
            ICNT ← ICNT +1
    End
    begin // main method counter
    MMCNT← 1
    Repeat till readline ≠ NULL do
    if readline has "String args[]) or String [] args)"
    then
            MMCNT← MMCNT +1
    end
end
where:
    -jp is java program.
    -cf is class found.
    -ccnt is class count.
    -mf is method found.
    -mcnt is method count.
    -acf is abstract class found.
    -accnt is abstract class count.
    -amf is abstract method found.
    -amcnt is abstract method count.
    -interf is interface found.
    -icnt is interface count.
    -mmf is main method found.
    -mmcnt is main method count.
```

B. Auto Contracts Generator pseudo code

```
performance←"";
fns←(ArrayList<String>)session.getAttribute("fns");
br ← null;
sno←1;
for(String fn:fns)
begin
   br ← new BufferedReader(new FileReader(DAOFactory.path1+fn));
   SCL←"";
   DecisionList←"";
   AutoContracts←"";
   ccnt←0;
   while ((SCL ← br.readLine()) !← null)
   begin

   if(SCL.contains("if")&&!SCL.contains("/")&&!SCL.contains("*"))
      begin
         DecisionList+ ← SCL+"<br>";
          ac ← "";
         if(SCL.contains("isValidForm"))
         begin
            ac ← GenerateAutoContracts.func(SCL);
         end
         else if(SCL.contains("intValue()"))
         begin
            ac ← GenerateAutoContracts.func(SCL);
         end
         else if(SCL.contains("delCustomerPayee"))
         begin
            ac ← GenerateAutoContracts.func(SCL);
         end
         else if(SCL.contains("addCustomerPayee"))
         begin
            ac ← GenerateAutoContracts.func(SCL);
         end
         else if(SCL.contains("checkDate"))
         begin
            ac ← GenerateAutoContracts.func(SCL);
         end
         else if(SCL.contains("updateCustomerPreference"))
         begin
            ac ← GenerateAutoContracts.func(SCL);
         end
         else if(SCL.contains("next()"))
         begin
            ac ← GenerateAutoContracts.func(SCL);
         end
         else if(SCL.contains("equals"))
         begin
            ac ← GenerateAutoContracts.funeq(SCL);
         end
         else
         begin
            ac ← GenerateAutoContracts.fune(SCL);
         end
         AutoContracts+ ← ac+"<br>";
         ccnt++;
      end
   end
   dlcnt←0;
   accnt←0;
   if(DecisionList.equalsIgnoreCase(""))
   begin
      DecisionList ← "No Decision List";
Dlcnt ← 0;
   end
   else
   begin
      String dcl[] ← DecisionList.split("<br>");
      Dlcnt ← dcl.length;
   end
   if(AutoContracts.equalsIgnoreCase(""))
   begin
      AutoContracts ← "No Auto Contracts";
      Accnt ← 0;
   end
   else
   begin
      acl[] ← AutoContracts.split("<br>");
      accnt ← acl.length;
   end
   performance+ ← "['"+fn+"', "+dlcnt+", "+accnt+"],";
   sno++;
end
```

VI. RESULTS AND DISCUSSION

Performance evaluation for sensitive classes of the software under test is performed using Object oriented metrics acquired from code analyser. The program code of each software version is analysed to obtain statistical information. Trace events provide information about behavioural dependencies between classes. Execution time and Frequency of every dependent class to arrive at the number of methods invoked are computed. Popularity rank that identifies the change prone classes is then determined.

TABLE II
SENSITIVE CLASSES PERFORMACE EVALUATION

Sensitive class name	Execution Time	Frequency	Popularity rank
DBConnection	Faster	2	2
Exception	Medium	8	1
HttpServlet	Faster	2	2

TABLE III
BACKGROUND METRICS

Background Metrics	Counts
1. Coupling between object classes (CBO):	26
2. Number of Children (NOC):	12
3. Number of Attributes (NOA):	376
4. Number of Instance Variable (NIV):	34
5. Depth of Inheritance Tree (DIT):	8
6. Number of Methods per Class(Min) (NOMMIN):	1
7. Number of Methods per Class(Max) (NOMMAX):	63
8. Number of Instance Method (NIM):	102
9. Number of Local Methods (NLM):	229
10. Response For a Class (RFC):	32
11. Number of Static Methods(NOSM):	1
12. Number of Private Methods (NPRM):	1
13. Number of Protected Methods (NPROM):	3
14. Number of Public Methods (NPM):	97
15. Lack of Cohesion amongst methods (LCOM):	6
16. Number of Class(NC):	26
17. Number of Lines of Source Codes(NLSC):	3481
18. Number of Iteration(NOI):	129
19. Number of called methods(NCM):	172
20. Rate of called methods(RCM):	8
21. Number of method invocations on a class(NMI):	172
22. Number of internal method invocations on a class(NIMI):	4
23. Number of external method invocations on a class(NEMI):	98
24. Number of called class (static) methods(NCCM):	1
25. Number of class (static) method calls on a class(NCMI):	0
26. Number of created instances (NCI):	38
27. Number of created objects by the class instances(NCO):	21
28. Total number of calls(TI):	172
29. Number of calls by Owner(ITI):	4
30. Number of calls by Foreign(ETI):	98

The above table shows the mandatory background metrics of the software application for 30 parameters. Our AutoAssertMine model takes into consideration the background metrics to easily identify the sensitive or dependent classes and its relationship with depending classes. But the GoldMine tool does not consider these metrics while generating the candidate assertions.

A. Auto Contracts Performances:

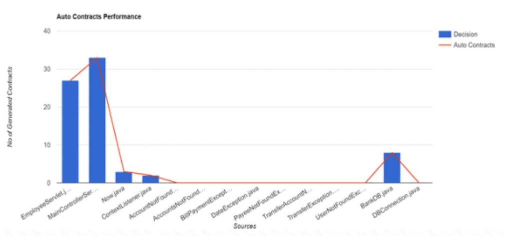

Figure 2 Automated contracts generated for Banking Input Software

TABLE IV
COMPARISON BETWEEN GOLDMINE TOOL VS. AUTOASSERTMINE

Source Files	AutoAsserMine Assertions	GoldMine Assertions
EmployeeServlet.java	27	9
MainControllerServlet.java	33	11
Now.java	3	1
ContextListener.java	2	1
AccountNotFoundException.java	0	0
AccountsNotFoundException.java	0	0
BillPaymentException.java	0	0
DateException.java	0	0
PayeeNotFoundException.java	0	0
TransferAccountNotFoundException.java	0	0
TransferException.java	0	0
UserNotFoundException.java	0	0
BankDB.java	8	3
DBConnection.java	0	0

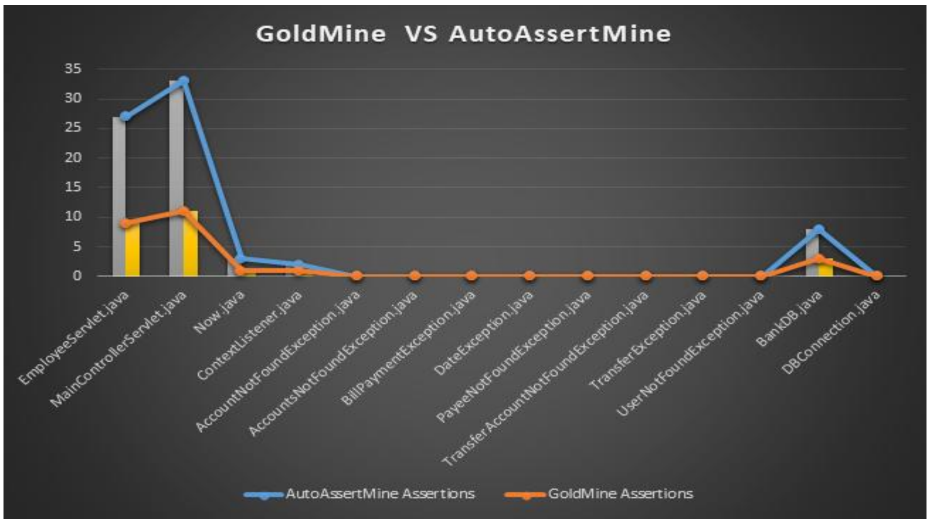

Figure 3. Performance graph of GoldMine tool and AutoAssertMine

The change proneness model and behavioral dependencies is evaluated. The number of assertions generated by the proposed AutoAssertMine method is found to be higher than those generated by GoldMine tool

VII. CONCLUSION

Existing methods of contract creations are reviewed and a simpler and lesser tedious approach to derive contracts automatically without manual interference of programmers is proposed. Further the generation of contracts through decision trees can aid in analysis of performance on particular methods and classes and in eliminating various programming errors including rare Concurrency related bugs. The proposed work focuses on mining the specifications to produce programming contracts aiming at higher number of bugs being identified. Optimizing the contracts forms the next step for future work on software verification in Time constrained setting.

REFERENCES

1. B. Meyer. Applying “design by contract”. Computer, 25(10):40–51, Oct. 1992.
2. Y. Cheon and A. Perumandla. Specifying and Checking Method Call Sequences of Java Programs. Software Quality Control, 15(1):7–25, Mar. 2007.
3. R.Demeyer and W.Vanhoof. Static Application-Level Race Detection in STM Haskell using Contracts. In Proc. of PLACCES. Open Publishing Association, 2013.
4. C. Hurlin. Specifying and checking protocols of multithreaded classes. In Proc. of SAC’09, pages 587–592. ACM, 2009.

5. D. G. Sousa, R. J. Dias, C. Ferreira, and J. M. Lourenço. Preventing atomicity violations with contracts. eprint arXiv:1505.02951, May 2015.
6. B. Meyer. Object-Oriented Software Construction, 2nd edition. Prentice Hall, 1997
7. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. SIGSOFT Softw. Eng. Notes, 31(3):1–38, 2006.
8. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In CASSIS 2004, volume 3362 of LNCS. Springer, 2004
9. Scott A. Carr, Francesco Logozzo, Mathias Payer. Automatic Contract Insertion with CCBot. IEEE Transactions on Software Engineering (Volume: 43, Issue: 8, Aug. 1 2017), Page(s): 701 – 714
10. Abdullah Alshanqiti.,Reiko Heckel., Extracting Visual Contracts from Java Programs (T). In Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on 9-13 Nov. 2015. INSPEC Accession Number: 15699219
11. Julien Signoles, Nikolai Kosmatov, & Kostyantyn Vorobyov. E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs. In An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (2017). Volume 3, 2017, Pages 164–173
12. Azra Shamim., Hameed Hussain., & Maqbool Uddin Shaikh.: A framework for generation of rules from decision tree and decision table. In Information and Emerging Technologies (ICIET), 2010 International Conference.
13. F. Logozzo, “Practical Specification and Verification with Code Contracts,” in Proceedings of the 2013 ACM SIGAda Annual Conference on High Integrity Language Technology, ser. HILT ’13. New York, NY, USA: ACM, 2013, pp. 7–8. [Online]. Available: http://doi.acm.org/10.1145/2527269.2534188
14. Schmidt, R. F.: Software engineering architecture-driven software development. Amsterdam: Elsevier: doi:https://doi.org/10.1016/B978-0-12-407768-3.00015-X, (2013)
15. Knutson, C., & Carmichael, S.: Verification and Validation. Embedded Systems Programming, Vol. 25 (2001).
16. Gabmeyer, S., Kaufmann, P., Seidl, M., Gogolla, M., & Kappel, G. A feature-based classification of formal verification techniques for software models. Software & Systems Modeling. (2017) 1-26.
17. Dias, R. J., Ferreira, C., Fiedor, J., Lourenço, J. M., Smrcka, A., Sousa, D. G., & Vojnar, T. Verifying Concurrent Programs Using Contracts. In Software Testing, Verification and Validation (ICST). (2017) 196-206.
18. Eslamimehr, M., Lesani, M., & Edwards, G.: Efficient Detection and Validation of Atomicity Violations in Concurrent Programs. Journal of Systems and Software. (2017).
19. e Silva, R. A. B., Arai, N. N., Burgareli, L. A., de Oliveira, J. M. P., & Pinto, J. S.: Formal verification with frama-C: A case study in the space software domain. IEEE Transactions on Reliability, Vol. 65(3), (2016) 1163-1179.
20. Podgurski A, Clarke LA. A formal model of program dependences and its implications for software testing, debugging, and maintenance. IEEE Transactions on Software Engineering 1990; 16(9): 965–979. DOI: 10.1109/32.58784.
21. Callo Arias TB, Spek P, Avgeriou P. A practice-driven systematic review of dependency analysis solutions. Empirical Software Engineering 2011; 16(5): 544–586. DOI: 10.1007/s10664-011-9158-8.
22. Shi, Q., Huang, J., Chen, Z., & Xu, B.: Verifying Synchronization for Atomicity Violation Fixing. IEEE Transactions on Software Engineering: Vol. 42(3) (2016) 280-296.
23. EtentingHub – Online Free Software Testing Tutorial. (n.d.). Software Testing Verification. Retrieved January 27, 2018, from http://www.etstnghub.com/testing_verification.php
24. Quinlan, J. R. (1987). "Simplifying decision trees". International Journal of Man-Machine Studies. 27 (3): 221. doi:10.1016/S0020-7373(87)80053-6.
25. Tong Zhang and Daniel Saab, Jacob A. Abraham: Automatic Assertion Generation for Simulation, Formal Verification and Emulation. 2017 IEEE Computer Society Annual Symposium on VLSI
26. Shobha Vasudevan, David Sheridan, Sanjay Patel, David Tcheng, Bill Tuohy, Daniel Johnson: GoldMine: Automatic assertion generation using data mining and static analysis. Published in 2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)