World Academy of Science, Engineering and Technology
International Journal of Computer and Systems Engineering
Vol:12, No:9, 2018

A Formal Property Verification for Aspect-Oriented Programs in Software Development

Moustapha Bande, Hakima Ould-Slimane, Hanifa Boucheneb

***Abstract*—**Software development for complex systems requires efficient and automatic tools that can be used to verify the satisfiability of some critical properties such as security ones. With the emergence of Aspect-Oriented Programming (AOP), considerable work has been done in order to better modularize the separation of concerns in the software design and implementation. The goal is to prevent the cross-cutting concerns to be scattered across the multiple modules of the program and tangled with other modules. One of the key challenges in the aspect-oriented programs is to be sure that all the pieces put together at the weaving time ensure the satisfiability of the overall system requirements.

Our paper focuses on this problem and proposes a formal property verification approach for a given property from the woven program. The approach is based on the control flow graph (CFG) of the woven program, and the use of a satisfiability modulo theories (SMT) solver to check whether each property (represented par one aspect) is satisfied or not once the weaving is done.

***Keywords*—**Aspect-oriented programming, control flow graph, satisfiability modulo theories, property verification.

I. Introduction

ASPECT-ORIENTED PROGRAMMING (AOP) [17] has emerged as a programming paradigm that aims to improve the separation of concerns in software development. In AOP, the system is divided into two parts: the base program containing the main functionality of the system and the aspect program that is composed by the cross-cutting functionality [10]. This technique has brought along with it new mechanisms and concepts for implementing crosscutting concerns in a modular manner. Among those crosscutting concerns, security requirements are considered to be more representative since they tend to scatter and tangle with other concerns of a software system. AspectJ [16] is the most popular AOP language and is based on the Java language. It adds the following new concepts to the Java language:

- **Aspect.** It is a module for handling cross-cutting concerns. It can be seen as a class-like construct.
- **Join point.** It is a well-defined point within a class where a concern is going to be attached during the execution of the program (e.g. method calls, exception thrown,...).
- **Advice.** It is the action taken by an aspect at a particular join point. An advice is implemented as a method of the aspect class. This method is executed *before* or *after* the join point is reached. It may also be executed *around* the join point.
- **Pointcut.** It is a group of join points that need to be matched before running an advice.
- **Weaving.**. It is the process of executing the relevant advice at each join point. This is achieved by a key component called the *aspect weaver*. This component takes the core modules and the aspects and then composes the final program also known as ”woven program”.

AOP technique helps developers to gain in modularity and ease to maintain because this approach isolates crosscutting concerns into aspects. However, in software testing, AOP brings along with it new issues [1] and summarized as follows.

- **Aspects identity and existence.** They do not have independent existence since they usually depend on the context of other classes and also the execution context.
- **Weaving time.** Aspects also tightly linked to the classes they are woven during the execution time. Any change to one of the these classes could have an impact on the aspects.
- **Data and control dependencies between aspects and classes.** During the weaving process, the resulting control flow nor the data flow structure of the woven program is not obvious for the developer.
- **Emergent behavior.** Many faults occur because of the implementation of a given class or aspect, or a side effect of the weaving process of multiple aspects.
- **Changes in normal and exceptional control flow**. Advices containing statements that possibly throw an exception might cause an implicit modification in the system control flow [7].

With the increasing size and complexity of software systems, most of software development teams need tools or techniques that may help them to detect errors or to find inconsistencies in the source code. Among the various functionality of a program, security modules are more critical because their violation may lead to enormous loss for enterprises. In AOP, it is not accurate to assume that the successful testing in isolation of the base program and each aspect implies the consistency of the overall system, since interactions and interference usually happen between them and changes in one aspect may have significant impacts on the whole execution of the AOP program.

The problem of detecting errors or inconsistencies during the weaving time has been variously tackled by aspect-oriented community. Some of the previous work have focused on

Moustapha Bande is with the Department of Computer Engineering, Ecole Polytechnique de Montreal, Montreal (Qc), Canada (e-mail: moustapha.bande@polymtl.ca).

Hakima Ould-Slimane is with the Department of Electrical Engineering, École de Technologie Supérieure, Montreal (Qc), Canada (e-mail: cc-hakima.ould-slimane@etsmtl.ca).

Hanifa Boucheneb is with the Department of Computer Engineering, Ecole Polytechnique de Montreal, Montreal (Qc), Canada (e-mail: hanifa.boucheneb@polymtl.ca).

finding approaches and tools for aspect-oriented program testing, using Unified Modeling Language (UML) models [20], [3], [2] or graph-based testing [12], [25], [5]. In this work, we present another approach that combines some of the existing approaches for verifying the satisfiability of properties covered in a woven aspect-oriented program. Every time, one critical aspect related to a specific property is added to the program, it may be possible to check if all the properties being covered by the program remain satisfied. Indeed, such critical requirements need to be handled with accuracy and only an automatic tool that can explore all the necessary states of the program execution could help to avoid the software misbehavior.

The main contribution of this paper is an automatic misbehavior detection of a given property at the weaving time. It is an integrated approach that uses the control flow graph (CFG) of the woven program and then, by transforming the derived CFG into Z3-SMT solver model, we can easily verify the satisfiability of any property. This work can be extended to any new added aspect and also used in collaborative software development in order to automatically verify whether the woven program still satisfies a given property.

The remainder of this paper is organized as follows. In Section II, we present our formal property verification approach for aspect-oriented programs. Section III deals with our model simulation results and Section IV is dedicated to some previous work in AOP software testing. Section V finally concludes the paper and gives some future work.

II. Formal Property Verification in AOP Programs

For our design purpose, we use an automatic teller machine (ATM) transaction management system.

A. Experimental Example: An ATM Transaction Management

As we can see, this example refers to an ATM transaction management in an AspectJ program.

The listing 1 shows the Java class *BankAccount* which simulates the functionality of a bank account and represents the *base program*. The aspect defined in Listing 2 is used to monitor the access to a bank account by a user. This aspect captures the authentication access control. We define another aspect in Listing 3 that captures transaction checking (e.g. being sure that an account has enough money for a withdrawal transaction). Our experimental source code example consists of one class (BankAccount) and two aspects (authentication and transaction checking).

- *Class BankAccount.* This class records user's account information: user name, account ID, pin code and account balance. It also defines methods such as withdrawal and deposit.
- *Aspect AspectAuthentication.* This aspect implements authentication goal. That is giving access to authorized users to their account information. The goal that keeps the information secret from unauthorized people is almost the most common one in information security. In the context of ATM transaction management, violation of such an information security property may be crucial for the bank reputation.
- *Aspect AspectChecking.* It implements a transaction requirement checking. We use a *before* advice to check whether a withdrawal transaction is possible or not considering the target account balance. It also includes a *after* advice that simply displays the target account balance after the transaction has been successfully completed.

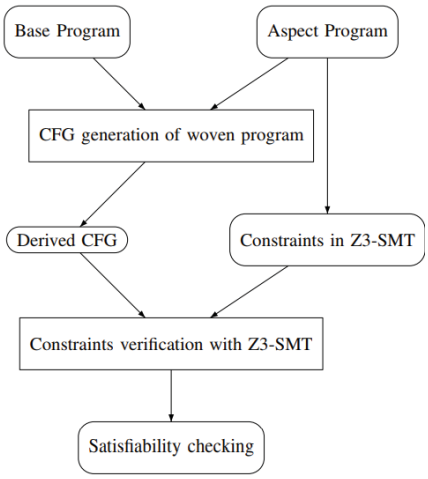

Fig. 1 Our approach design

Given an aspect-oriented program and any property that may be verified through the execution of this program, we build a fully integrated approach that verifies if the property is satisfied or not during the weaving time.

B. Our Design Approach

Our framework in Fig. 1 uses an existing algorithm that aims to generate the control flow graph of the program and a formal system verification tool that is used to derive a SMT solver from the generated CFG and also to verify that some properties can be automatically verified.

For our design purpose, we use the algorithm proposed by [21] to generate the CFG of our program. In their work, the authors implemented a tool called *AJcFgraph Builder* that automatically derives an aspect-oriented control flow graph (AOCFG) giving an AOP program. Based on their algorithm, Fig. 2 shows the CFG of our AOP program and displays the case an access is granted to a user account for doing his transactions. In order to make the graph more readable, we annotate the nodes with the character B followed by the line number of the source code for the *Base* program and A followed by the line number of the source code for the *AspectAuthentication* program and C for *AspectChecking* program . The small b added to the node $B78b$ and $B79b$ stands for *before* advice, whereas small a added to node $B79a$ is used for *after* advice. We also add (in yellow color) nodes and edges that denote the case where the pin code is not valid ($A8 \implies A11 \implies A12$) and the one where account balance is less than the withdrawal amount ($C7 \implies C8 \implies C9$).

Listing 1: Class BankAccount

```
1
2  package example;
3  import example.BankAccount;
4  import java.io.*;
5  import java.util.Properties;
6
7  public class BankAccount
8  {
9      private String accountOwner;
10     private String accountNumber;
11     private String accountPin;
12     private double balance;
13
14     public BankAccount(String owner, String account, String pin, double balance) // BankAccount constructor
15     {
16         this.accountOwner = owner;
17         this.accountNumber = account;
18         this.accountPin = pin;
19         if (balance > 0.0)
20             this.balance = balance;
21     }
22
23     public void deposit(double depositAmount)  // method for deposits
24     {
25       balance = balance + depositAmount; //update the balance
26     }
27
28     public void withdrawal(double withdrawalAmount) // method withdrawal
29     {
30       balance = balance - withdrawalAmount; // update the balance
31     }
32
33     public double getBalance()
34     {
35         return balance;
36     }
37
38     public void setAccount(String account)
39     {
40         this.accountNumber = account;
41     }
42
43     public void setOwner(String owner)
44     {
45         this.accountOwner = owner;
46     }
47     public void setPin(String pin)
48     {
49         this.accountPin = pin;
50     }
51     public String getOwner()
52     {
53         return accountOwner;
54     }
55
56     public String getAccount()
57     {
58         return accountNumber;
59     }
60     public String getPin()
61     {
62         return accountPin;
63     }
64
65     public boolean authenticate(String userName, String pin) throws Exception{
66
67         File userFile = new File("D:/eclipse/workspace/users.txt");
68         FileInputStream in = new FileInputStream(userFile);
69         Properties properties = new Properties();
70         properties.load(in);
71         String storedPassword = properties.getProperty(userName);
72         in.close();
73         return pin.equals(storedPassword);
74     }
75
76     public static void main(String[] args)
77     {
78       BankAccount account = new BankAccount("Toto","120541263","789",500);
79       account.deposit(500);
80       account.withdrawal(300);
81       }
82 }
```

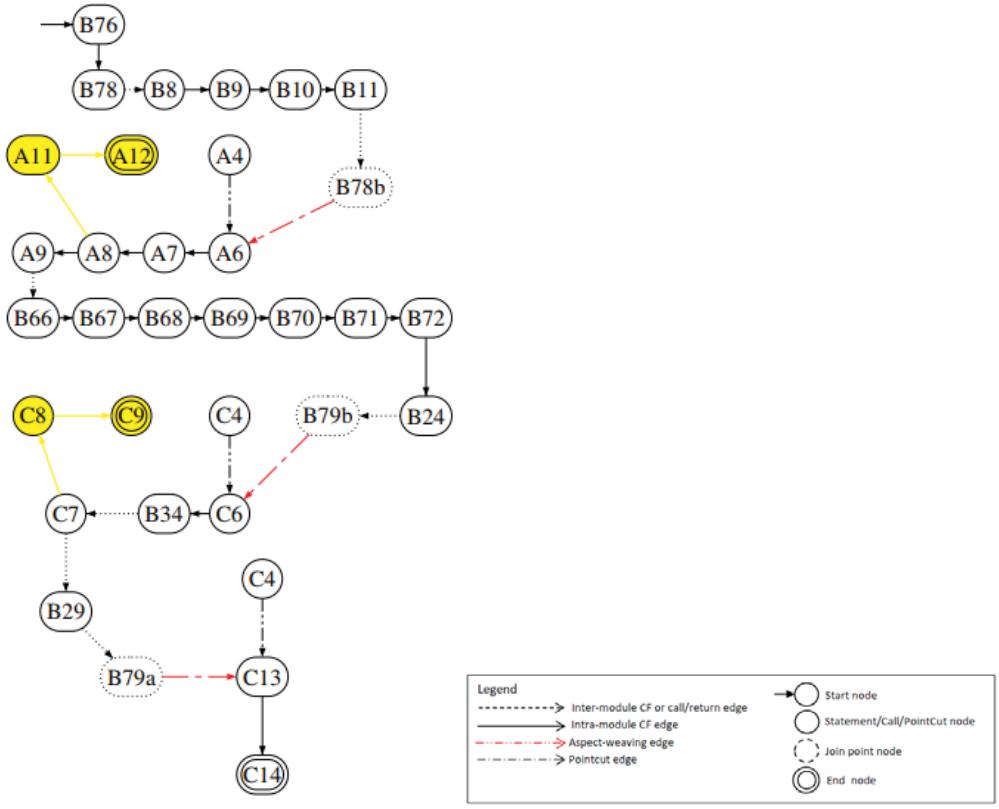

Fig. 2 The derived CFG of our program

Listing 2: Aspect user's authentication

```
package org.example.aop;
import example.BankAccount;
public aspect AspectAuthentication {
    pointcut authentication (BankAccount b, double x): (call(* BankAccount.withdrawal(..)) || call(* BankAccount.deposit(..))) && target(b) && args(x);

    before(BankAccount b, double x) : authentication(b,x) {
        try {
            if (!b.authenticate(b.getOwner(),b.getPin()))
            {
            System.out.println("Access denied ... Incorrect pin");
            System.exit(0);
            }
        } catch (Exception e) {
            System.out.println("Error ... ");
        }
    }
}
```

Listing 3: Aspect transaction checking

```
package org.example.aop;
import example.BankAccount;
public aspect AspectChecking {
    pointcut checking(BankAccount b, double x): call(* BankAccount.withdrawal(..)) && target(b) && args(x);

    before(BankAccount b, double x) : checking(b, x) {
        if(b.getBalance() < x) {
            System.out.println("Sorry Your account does not have. $" + x + " to withdrawn ...");
            System.exit(0);
        }
    }

    after(BankAccount b, double x): checking(b, x) {
        System.out.println("Transaction successfully done!!! Your current balance is : $" + b.getBalance());
    }
}
```

After the generation of the CFG graph, we build our model by deriving Z3 SMT model from the CFG graph.

C. Our Z3 Model Design

Satisfiability Modulo Theories (SMT) is the Satisfiability of formulas with respect to some background theory [4]. According to the authors in [8], Z3 is an new SMT solver tool from Microsoft Research. This tool aims at checking the satisfiability of logical formulas over one or more theories and is based on first-order logic. A given formula ϕ is said to be *satisfiable* if there exists an interpretation that makes ϕ true. A formula is said to be *valide* if it is true under all structures. For example, $x + y > 3$ is satisfiable under the interpretation $x \mapsto 1$ and $y \mapsto 3$. We report in this paper only some relevant Z3 constructs that we use in our model due to the limit of the paper length. For our model design purpose, we use $Z3Py$, a Python interface for Z3 solver.

- *Sorts.* Z3 allows users to declare new data types called sorts (e.g. $a = DeclareSort('a')$).
- *Constants.* They are functions that take no argument (e.g. $b = Const('b', a)$).
- *Quantifiers.* Universal quantifier is represented by $ForAll$ and existential one by $Exists$.
- *Solver().* We can use it to create a given solver instance (e.g. $s = Solver()$).
- *Add().* We can use this method to add constraints to the solver (e.g. $s.add(x <= 8)$)
- *Check().* We can call this method to check the satisfiability of all constraints that are associated to the solver (e.g. $s.check()$) and the result returned is either sat (satisfiable), or $unsat$ (unsatisfiable).

In order to derive the Z3 model from the CFG of the woven program, we use the rules summarized in Table I.

- We declare in $Z3Py$ a graph that is a representation of the CFG derived from the woven program. We then add the following attributes to nodes and edges to capture data information during the simulation of the model:
  - *Node*: this attribute captures the name of a node in the graph.
  - *ExecStatus*: this attribute captures the execution status of an edge in the graph.
  - *isJoinPoint*: this attribute return true if the node is a join point node and false otherwise.
  - *isAspectWeavingEdge*: this attribute return true if the edge is an aspect-weaving edge and false otherwise.
  - *isPointcutEdge*: this attribute return true if the edge is an pointcut edge and false otherwise.
- We define the function $isAuth(u, p)$ that returns $true$ if the user u enters a valid pin p and $false$ otherwise, giving the attributes of the edge and nodes being traversed in the graph. This corresponds to the advice of the aspect program. It is declared using $isAuth = Function('isAuth', IntSort(), IntSort(), BoolSort())$.
- For our simulation purpose, we define an array variable that contains 3-tuple of user, his pin code and his account balance. In this array variable, are stored random generated triples of user, pin and balance.

```
====== RESTART: D:/z3-4.6.0-x64-win
/bin/python/Cfg_verification_ATM.py
======
sat
B76 --> B78
B78 --> B8
B8 --> B9
B9 --> B10
B10 --> B11
B11 --> B78b
B78b --> A6
A4 --> A6
A6 --> A7
A7 --> A8
A8 --> A11
A11 --> A12
>>>
```

Fig. 3 The model checking output with constraint (1)

III. Properties Verification

We recall that the goal of our model is to automatically verify properties for an AOP program. Based on the parameters used to run the main method, we verify whether an unauthorized user could access or not the ATM system. After building our Z3 model, we add some constraints into it and then check its satisfiability.

First of all, we add the constraint

$$s.add(ForAll([u_i, p_i], Implies(Not(isAuth(u_i, p_i)), g.es[11]["ExecStatus"]))). \quad (1)$$

The checking of the solver using $print(s.check())$ outputs the result Sat meaning that our model is *satisfiable*. This shows that for any given user u_i and his pin code p_i, if the pin code is not valid then we do have an edge between nodes $A11$ and $A12$. The attribute *"ExecStatus"* of this edge is then *True* corresponding to the edge whose index is 11. Consequently, the access is denied and the end node is $A12$. We output only the part of the graph with edges having their execution status at *true* in Fig. 3.

We also add to the solver this constraint

$$s.add(ForAll([u_i, p_i, b_i, x], Implies(And(isAuth(pi, fi), x <= b_i), g.es[32]["ExecStatus"]))). \quad (2)$$

Then, we run the model using $check()$ method and the output result of the model checking is Sat, that is the user u_i has entered a valid code pin and the withdrawal amount is less than his account balance, so that the transaction is successfully completed. Thus, the attribute *"ExecStatus"* of the edge whose index is 32 is *True*. Consequently, the end node in the output in Fig. 4 is $C14$.

Finally, We add to the solver the following constraint that verifies whether the user do not enough money to make his withdrawal transaction.

$$s.add(ForAll([u_i, p_i, b_i, x], Implies(And(isAuth(pi, fi), x > b_i), g.es[27]["ExecStatus"]))). \quad (3)$$

Then, we run the model using $check()$ method and the output result of the model checking is Sat, that is, the user has access to the ATM system but his account balance is not enough to make his withdrawal transaction. Thus, the

TABLE I
FROM CFG TO Z3 - TRANSFORMATION RULES

	CFG	Z3
1	Set of nodes Nd	Nd = DeclareSort('Nd')
2	A node $nd1 \in Nd$	$nd1$ = Const('nd1',Nd)
3	Set of edges Eg	Nd = DeclareSort('Eg')
4	An edge $eg(nd_i, nd_j) \in Nd \times Nd$	ed = Function('ed', Nd, Nd, BoolSort())
5	$isPointCut(nd_i)$	isPointCut = Function('isPointCut', Nd, BoolSort())
6	$isWeavingEdge(nd_i, nd_j)$	isWeavingEdge = Function('isWeavingEdge', Nd, Nd, BoolSort())
7	$isJointPointEdge(nd_i, nd_j)$	isJointPointEdge = Function('isJointPointEdge', Nd, Nd, BoolSort())

```
====== RESTART: D:/z3-4.6.0-x64-win/bin
/python/Cfg_verification_ATM.py ======
sat
B76 --> B78
B78 --> B8
B8 --> B9
B9 --> B10
B10 --> B11
B11 --> B78b
B78b --> A6
A4 --> A6
A6 --> A7
A7 --> A8
A8 --> A11
A11 --> A12
A8 --> A9
A9 --> B66
B66 --> B67
B67 --> B68
B68 --> B69
B69 --> B70
B70 --> B71
B71 --> B72
B72 --> B24
B24 --> B79b
B79b --> C6
C4 --> C6
C6 --> B34
B34 --> C7
C7 --> C8
C8 --> C9
C7 --> B29
B29 --> B79a
B79a --> C13
C4 --> C13
C13 --> C14
>>>
```

Fig. 4 The model checking output with constraint (2)

```
====== RESTART: D:/z3-4.6.0-x64-win
/bin/python/Cfg_verification_ATM.py
======
sat
B76 --> B78
B78 --> B8
B8 --> B9
B9 --> B10
B10 --> B11
B11 --> B78b
B78b --> A6
A4 --> A6
A6 --> A7
A7 --> A8
A8 --> A11
A11 --> A12
A8 --> A9
A9 --> B66
B66 --> B67
B67 --> B68
B68 --> B69
B69 --> B70
B70 --> B71
B71 --> B72
B72 --> B24
B24 --> B79b
B79b --> C6
C4 --> C6
C6 --> B34
B34 --> C7
C7 --> C8
C8 --> C9
>>>
```

Fig. 5 The model checking output with constraint (3)

attribute *"ExecStatus"* of the edge whose index is 27 is *True*. Consequently, the end node in the output in Fig. 5 is $C9$.

We note that the increase of the number of users does not affect the size of our model since the system follows the execution of the $main()$ method. In addition, it is also possible to define other types of properties and add them to the model and let Z3-SMT solver checks the satisfiability of the whole system. In Z3, it is possible to handle function composition, that is a function call in the CFG that also calls other functions. We could also add this kind of constraint to the model and check the solver satisfiability. That helps to make modular verification and also incremental one. Our model is flexible and could be used whenever a developer adds a new aspect that captures a specific concern. It can be extended to a collaborative environment where developers together add components in a base program.

IV. RELATED WORK

Many researches have focused on software testing using several formal methods and tools to detect faults and errors during compilation or running time. These approaches can be classified into three categories:

The first category is related to those approaches that use model checking to verify whether a given property is verified or not. Among them, in [13], the authors proposed an aspect-oriented Petri net with Aspect-Oriented Software Development (AOSD) mechanisms. They developed an automated approach for formally analyzing the software design using the model checking technique and analysis tool PROD. In [27], Xu et al. presented a framework called Model-based Aspect/class Checking and Testing (MACT) that aims to test the conformance of aspect-oriented programs against their aspect-oriented state model. The framework also uses model checking for test generation from counterexamples. Fradet et al. [11] proposed a formal framework to enforce availability properties on services sharing resources. The authors used timed automata in order to express and enforce properties on execution time. In [24], they defined an automatic verification approach using model checking that verifies the correctness of AOP-based programs. In order to check the correctness of the woven program, they defined a

property in Computation Tree Logic (CTL) and represent this property by an aspect that is finally used for the property verification. In [9], Denaro et al. proposed an approach based on analyzing aspect-based software components in order to verify safety properties. The aspects are modeled using a PROMELA process template and the derived model is analyzed with the model checker SPIN in order to verify the deadlock problem of the synchronization policy. Another state-based approach has been proposed in [26] for modeling the specification of an aspect-oriented design. In this paper, the authors used a Labeled Transition System Analyzer (LTSA) model checker to verify the generated finite state processes against the desired properties. In [22], a runtime verification framework for Java programs has been proposed. They instrumented Java bytecode with Linear-time Temporal Logic (LTL) formula and for the properties verification purpose, they used an automaton-based approach where transitions are implemented and triggered using aspects. They also developed a prototype called Java Logical Observer (JLO) that gives a way to derive a verification aspect from the formula. The technique used in this paper, however, implies injecting aspects in order to verify a specific property. That can become more challenging while the number of properties being verified grows. These papers did not, however, describe a modular verification methodology or most of them use source code instrumentation for the verification purposes.

The second category of related work is data-flow or control-flow based testing approaches. In [18], a pointcut-based coverage analysis approach using structural model based on Java bytecode has been proposed. In their approach, the authors used a control and data flow graph for their testing criteria. They also proposed a series of control flow and data flow testing criteria based on a PointCut-based Def-Use (PCDU) graph. However, they did not mention the data flow between integrated units among their testing criteria. In [28], the author proposed a data-flow-based unit testing approach in which three levels of testing are performed: intra-module, inter-module and inter-aspect or inter-class testing. The model is based on modeling a control graph flow for only the program classes and is extended to the woven program. Finally, the program issues are analyzed by computing def-use pairs of a class or an aspect. This work, however, does not focus on the advice interactions. Fradet et al. [11] proposed a formal framework to enforce availability properties on services sharing resources. The authors used timed automata in order to express and enforce properties on execution time for preventing denial of service attacks. However, they did not consider any change that may occur in the source code. In [19], a derivation of control flow and data flow based testing criteria for aspect-oriented programs has been defined. This model is used to support structural testing to unit testing of AspectJ programs. The key problem with existing dataflow test criteria is the difficulty in covering all types of data flow interactions for AOP programs. Harlem et al. [14] proposed an aspect-oriented declarative security policy specification language for in-lined reference monitoring. This language called SPOX (Security Policy XML) is an XML-based security policy specification in which policies denote an aspect-oriented security automaton.

In the first category of the aforementioned work, the authors instrumented the source code by injecting pieces of code (e.g. aspects) in order to verify a specific property. In the second category, testing criteria are defined on control flow without taking into account the data flow or the core program and the aspects are tested separately. Unlike the above work, our paper focuses on verifying a security property (e.g. authentication, privacy, etc.) encapsulated in an aspect for a woven program and also captures the interactions between the base program and the aspects.

The third category is UML-based transformation using sequence diagrams. Hossain et al. [15] proposed a transformation technique called ”Zigzag transformation” to introduce aspects at the architecture level in order to verify properties of the old architecture versus the ones introduced by the aspects. Bowles et al. [6] introduced a novel formal automated technique for weaving aspects using Z3-SMT solver. Their technique is based on UML sequence diagrams to capture the base and the advice and pointcut models are transformed into equivalent representations using Labelled Event Structures (LES). They also compared the performance using Z3 with their earlier work that used Alloy [6] for sequence diagram composition. Tahara et al. [23] proposed a tool called ”CAMPer” that uses Maude specification language to express dynamic aspect weaving. This tool takes an input UML class diagram, sequence diagrams and a Linear Temporal Logic (LTL) formula and automatically verifies the models.

Unlike the aforementioned work, our approach combines control flow graph generation and a SMT solver to check the satisfiability of properties by deriving a Z3 SMT model from the CFG.

V. Conclusion and Future Work

In this paper, a formal verification of properties in aspect-oriented programming is proposed. The main idea of our approach is to generate a control flow graph from a giving woven aspect-oriented program and to derive Z3-SMT model in order to verify some critical properties such as confidentiality. Our approach is flexible and can be extended to both AOP programs with many aspects that encapsulate security property or specific requirement and collaborative software development environments where many developers can collaborate in the same project. In such an environment, this formal verification technique could be used to verify critical properties whenever any developer adds or modifies a piece of code (e.g. aspects, base program).

As future work, we are planning to extend this work to a collaborative software development including dealing with many aspects. Finally, we will look into building a tool that can automatically generate Z3-SMT model from the CFG.

References

[1] Roger T Alexander, James M Bieman, and Anneliese A Andrews. Towards the systematic testing of aspect-oriented programs. *Rapport technique, Colorado State University*, 2004.

[2] Chitra Babu and Harshini Ramnath Krishnan. Fault model and test-case generation for the composition of aspects. *ACM SIGSOFT Software Engineering Notes*, 34(1):1–6, 2009.

[3] Mourad Badri, Linda Badri, and Maxime Bourque-Fortin. Generating unit test sequences for aspect-oriented programs: towards a formal approach using uml state diagrams. In *Information and Communications Technology, 2005. Enabling Technologies for the New Knowledge Society: ITI 3rd International Conference on*, pages 237–253. IEEE, 2005.

[4] Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, Cesare Tinelli, et al. Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885, 2009.

[5] Mario L Bernardi. Reverse engineering of aspect oriented systems to support their comprehension, evolution, testing and assessment. In *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pages 290–293. IEEE, 2008.

[6] Juliana KF Bowles, Behzad Bordbar, and Mohammed Alwanain. Weaving true-concurrent aspects using constraint solvers. In *Application of Concurrency to System Design (ACSD), 2016 16th International Conference on*, pages 35–44. IEEE, 2016.

[7] Mariano Ceccato, Paolo Tonella, and Filippo Ricca. Is aop code easier or harder to test than oop code. In *First Workshop on Testing Aspect-Oriented Program (WTAOP), Chicago, Illinois*, 2005.

[8] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[9] Giovanni Denaro and Mattia Monga. An experience on verification of aspect properties. In *Proceedings of the 4th international workshop on Principles of software evolution*, pages 186–189. ACM, 2001.

[10] Tzilla Elrad, Mehmet Aksit, Gregor Kiczales, Karl Lieberherr, and Harold Ossher. Discussing aspects of aop. *Communications of the ACM*, 44(10):33–38, 2001.

[11] Pascal Fradet and Stéphane Hong Tuan Ha. Aspects of availability: Enforcing timed properties to prevent denial of service. *Science of Computer Programming*, 75(7):516–542, 2010.

[12] Ivan Gustavo Franchin, Otávio Augusto Lazzarini Lemos, and Paulo Cesar Masiero. Pairwise structural testing of object and aspect-oriented java programs. In *The 21th Software Engineering Brazilian Symposium, Joao Pessoa, PB, Brazil*, 2007.

[13] Yujian Fu, Junhua Ding, and Phil Bording. An approach for modeling and analyzing crosscutting concerns. In *Service Operations, Logistics and Informatics, 2009. SOLI'09. IEEE/INFORMS International Conference on*, pages 91–97. IEEE, 2009.

[14] Kevin W Hamlen and Micah Jones. Aspect-oriented in-lined reference monitors. In *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, pages 11–20. ACM, 2008.

[15] Md Nour Hossain, Wolfram Kahl, and Tom Maibaum. A graph transformation approach to introducing aspects into software architectures.

[16] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An overview of aspectj. In *ECOOP 2001Object-Oriented Programming*, pages 327–354. Springer, 2001.

[17] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-oriented programming*. Springer, 1997.

[18] Otávio Augusto Lazzarini Lemos and Paulo Cesar Masiero. A pointcut-based coverage analysis approach for aspect-oriented programs. *Information Sciences*, 181(13):2721–2746, 2011.

[19] Otávio Augusto Lazzarini Lemos, Auri Marcelo Rizzo Vincenzi, José Carlos Maldonado, and Paulo Cesar Masiero. Control and data flow structural testing criteria for aspect-oriented programs. *Journal of Systems and Software*, 80(6):862–882, 2007.

[20] Philippe Massicotte, Mourad Badri, and Linda Badri. Generating aspects-classes integration testing sequences a collaboration diagram based strategy. In *Software Engineering Research, Management and Applications, 2005. Third ACIS International Conference on*, pages 30–37. IEEE, 2005.

[21] Reza Meimandi Parizi and Abdul Azim Abdul Ghani. Ajcfgraph-aspectj control flow graph builder for aspect-oriented software. *International Journal of Computer Science*, 3:170–181, 2008.

[22] Volker Stolz and Eric Bodden. Temporal assertions using aspectj. *Electronic Notes in Theoretical Computer Science*, 144(4):109–124, 2006.

[23] Yasuyuki Tahara, Akihiko Ohsuga, and Shinichi Honiden. Formal verification of dynamic evolution processes of uml models using aspects. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2017 IEEE/ACM 12th International Symposium on*, pages 152–162. IEEE, 2017.

[24] Naoyasu Ubayashi and Tetsuo Tamai. Aspect-oriented programming with model checking. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 148–154. ACM, 2002.

[25] Fadi Wedyan and Sudipto Ghosh. A dataflow testing approach for aspect-oriented programs. In *High-Assurance Systems Engineering (HASE), 2010 IEEE 12th International Symposium on*, pages 64–73. IEEE, 2010.

[26] Dianxiang Xu, Izzat Alsmadi, and Weifeng Xu. Model checking aspect-oriented design specification. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, volume 1, pages 491–500. IEEE, 2007.

[27] Dianxiang Xu, Omar El-Ariss, Weifeng Xu, and Linzhang Wang. Testing aspect-oriented programs with finite state machines. *Software Testing, Verification and Reliability*, 22(4):267–293, 2012.

[28] Jianjun Zhao. Data-flow-based unit testing of aspect-oriented programs. In *Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International*, pages 188–197. IEEE, 2003.