An Automatable Framework for Formal Specification & Verification of Aspect Oriented Programs

M. Nafees Qamar[1 *], Aamer Nadeem[2], M. Uzair Khan[2]

[1]COMSATS Institute of Information Technology,
Islamabad, Pakistan.
[2]Center for Software Dependability,
Muhammad Ali Jinnah University,
Islamabad, Pakistan.
[1]{nafees_qamar@comsats.edu.pk}, [2]{anadeem, uzair@jinnah.edu.pk}

Abstract. Formal verification of software is a perennial problem and an inclusion to formal verification challenges is AOP (Aspect Oriented programs). Likewise, several contrary formal verification approaches exist for OOP (Object Oriented programs) but unable to deliver same robustness for AOP verification and also the proposed approaches to verify AOP seem to be less effective because of their adoptability for large systems. Although we believe that the potential solution would be to choose best existing formal methods for formal verification of AOP to avoid unnecessary additions and modifications in current state of the art. This paper gives a thorough survey of the existing work. The paper presents a holistic framework for formal verification of AOP by formally specifying aspects and classes, autonomously. Then the obtained specifications are integrated to get filtered definition of the AOP, i.e., merging the aspect specification with the class specification to preserve execution order of an AO program. Besides, to verify augmented system, the derivation of classes through de-compilation from compiled AOP is attained for generating their corresponding formal specifications. Former and latter generated formal specifications are compared to analyze the intended and exotic behavior of aspects. This helps us to ensure that the actual behavior of the system is according to the one specified. Additionally, our approach helps to identify ambiguities and contradictions present in the compiled AOP, and allows to eliminate them. We use Object-Z to write classes and propose our own notations and constructs for describing aspects and for formulating filtered formal specification of classes and aspects using the notion of filters from RTOZ (Real Time Object-Z*). A possible tool support realizes the automation process of the framework.

---

*is a senior member of Center for Software Dependability and presently working as Research Associate at COMSATS Institute of Information Technology, Islamabad.

1 Introduction

Aspect-Oriented Paradigm (AOP) promises to deliver robust software modularization and reusability. In this regard, AOP [11], [15] presents an acceptable solution to code scattering and tangling as it focuses on separation of concerns to avoid these troubles by treating core and crosscutting concerns separately at design level. The implementation of core concerns is language independent, while crosscutting concerns are written using any aspect based language, e.g., AspectJ [18], AspectC++ [17], etc. The main constructs in AOP are join points, pointcuts, advices, introductions, and aspects. An aspect represents a crosscutting concern and is weaved with one or more core concerns. The aspect code observes the base program and when certain pointcuts reach, the aspect code is weaved into there. Although AOP seems promising to avoid code scattering and tangling issues but correctness can not be guaranteed. AOP has conceptual differences which are more demanding than all the existing ones. Formal verification is a highly reliable solution to ensure the correctness of the software. We just need to use it in a productive manner.

This paper presents a novel framework for formal verification of AOP. The proposed approach autonomously tackles aspects and classes for formal specifications, and filters their formal specifications to form a fused formal specification of the complete AOP before actual weaving process and preserving execution order. On the other hand, the byte code[1] of augmented system is decompiled to obtain formulated (by weaver) classes, which is then transformed into the formal specifications by using derived classes. These formal specifications of derived classes are then verified against the formerly weaved formal specifications of classes and aspects. Although aspects are not realized as stand alone entity (except for their usability), hence their filtration is mandatory to visualize static definition of AOP. For this, we believe that the stand alone specifications of aspects and classes should be integrated into a single filtered specification, where all the constructs of the aspects are integrated with the classes. Groundwork for tool architecture depicts the automation process of the framework. Our contributions towards verification of AO programs are as follows: (i) a framework for static formal definition of classes & aspects, and how their conformance can be analyzed against weaved artifacts, (ii) identification of changes made by aspects into the core concerns, (iii) analysis of aspects with respect to their earlier specified behavior. (iv) a way of integrating formal specifications of aspects and classes, (v) introducing some new notations and constructs to elaborate AOP behavior, (vi) in the end, a way to cope with verification of static definition of the system against weaved system, and a possible tool support.

The proposed framework uses the theorem proving strategy to deal with verification of AO programs in a robust way; for instance, we have HOL-Z [24] implementation for Z schemas, consequently we may adopt it to verify the AOP. Several other solutions exist on the subject of de-compilation of the byte code [26], [27].Theorem proving techniques, for example [25], [29], [30], are powerful enough to deal with the verification issues. Regarding the automation of Z schemas, until now, a number of issues have been identified and as a result, robust notations and constructs exist for the mentioned problems [24], [30], [28].

[1]We are assuming a JAVA based system.

We have put forward some research challenges regarding our approach as this is a major undertaking, and before these tasks to carry out, a reasonable confidence is essential for accomplishing the absolute framework stages until it gets recognition.

The rest of this paper is organized as follows: Section 2 discusses related work, section 3 presents our approach for verification of AO programs along with an illustrative example, section 4 discusses possible tool support, while section 5 describes future directions and concludes the paper.

2 Related Work

Mousavi et al. [6] discusses the main characteristics of an aspect-oriented formal specification framework, which is based on a multiset transformation language called GAMMA. Their work intends to facilitate the specification-driven design, enabling formal validation, and design reuse at the requirement specification level. They provide a formal design of a small number of aspects, mainly related to distributed real-time systems. These aspects are kept separate and abstract from each other Hence a formal foundation of Aspect-Oriented specification and refinement towards implementation is laid.

Larsson and Alexandersson [5] discuss the formal verification of aspects that are written for fault tolerance. However, their work does not provide any concrete example to demonstrate their technique, and hence lacks in this perspective. In case of Aspect-Oriented programs, intricate part of verification is inclusion of aspects to core concerns. Generally in static verification, we analyze whether program satisfies the requirements specification. Formal methods can be excellent incentive to diagnose augmented system, as harmful aspects are always there. Weston et al. [8] talks about the modular verification of the aspects (as they are weaved with core concerns) using aspect tagging and data flow analysis of Control Flow Graphs (CFG).

Katz [4] presents a comprehensive survey of static analysis techniques of aspects and their verification approaches. Sakurail et al. [12] realization is possible of any aspect instance as the context of an advice execution as their interactions affects the system, while modifications are not apparent, and easily detectable. The composition, reuse, and interaction of stateful aspects necessitate the need of new testing techniques [10].

Storzer [13] reveals the issue that whether any aspect works as intended or not. Their work analyzes the impact of such aspects. By trace analysis, the behavior of program flow is determined. Some of the limitations of formal methods i.e., verification is costly [13], and verification implementation is restricted to some situations. Ubayashi [3] illustrates verification of aspects using model checking to see unexpected behavior of the system, such as deadlocks. Constructs of aspect-oriented programs, e.g., pointcuts, join points, advices, and introduction, are dynamic in nature, which enforce us to specify their behaviors.

Okun and Black [14] present model checking as a “light-weight” formal method to check the truth (or falsity) of statements. Their observations stress that more robust techniques are required to show the propagation of faults in the output. Denaro et al. [2] describes the suitability of aspect code for application of formal verification

techniques. In their approach, they initially select some general properties, which are relevant to a concern of aspect-oriented software, e.g. deadlock [3], [2], and then any formal method (for instance model checking) is selected for formal verification. The selection of formal method is done considering its suitability for the selected properties. Then the aspect, which is encapsulating the relevant code, is identified (for example, aspect coding the synchronization policy). It applies any formal technique after derivation of verification model. As per required, specification of the selected properties is exercised with proper formalism. In the end, their method report on the success of the verification or analyze the provided counter examples, e.g., invalid execution leading to deadlocks. Static analysis is usually manual or automated, but industry practitioners like Hailpern and Santhanam [7], describe that formal verification is used routinely by only small pockets of the industrial software community, particularly in the areas of protocol verification and embedded systems. Solutions to small chunks of problems are focused [1], [12], [10], [13], [3], [9], [20]. Therefore we need to develop a rigorous and automated solution for large scale problems to cope with new imposed challenges of AO programs.

Diversity of work also exists in de-compilation [26], [27], theorem provers [29], [25], Z-Schemas automation issues and writing formal specifications from the source code [31], [32].

3 The Proposed Framework

Our proposed approach is presented in Figure 1. Primarily core concerns and aspects

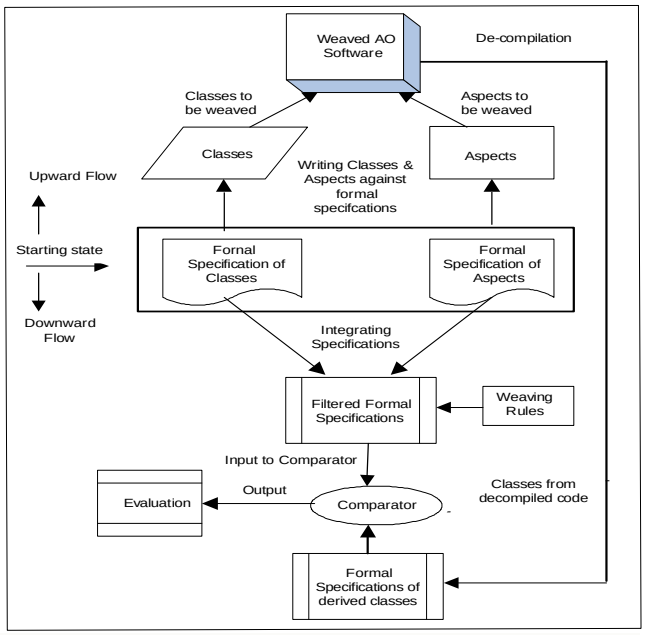

Fig. 1. Framework for verification of AO programs.

(crosscutting concerns) are specified using formal constructs (i.e., Object-Z for classes and our own notations for aspects). Then the separate formal specifications are combined to result in one filtered specification of the overall AO program. There are two main reasons behind filtration of these specifications. Firstly, by filtering formal specifications of aspects and classes together, we may be able to identify the contradictions and ambiguities in core and crosscutting concerns. Secondly, we can use them to verify against the formal specifications of the actual weaved code. Our proposed framework is divided into three main subsections which are (i) formal Specifications of aspects and objects and then filtration of these formal specifications (ii) construction of formal specifications through available byte code by de-compilation (which will be presented later) (iii) A verifier, which identifies conflicts (according to purported proves of the aspects and required services of the objects) between (i) and (ii). We employ Object-Z [16] constructs to write formal specifications of classes, and our own notations. Aspects are specialized in constructs, e.g., pointcuts, join points, advices, introduction etc., to facilitate description of crosscutting features, and are formally specified to describe semantics of aspects. Perhaps when the number of classes and aspects grow, the strategy will help to determine the interdependencies among the (i) aspects itself and (ii) in case of weaving them with the classes. Their conflict resolution can also become less problematic as it will also allow reasoning before compilation. We take an example of an AO program where we want to log the activities in a Stack class. Our example manifests the one part of the framework i.e., formal specification of aspect and class as well as the filtered specification of both. The following subsections illustrate the approach.

3.1 Formal Specifications of Core Concerns using Object-Z

A rich set of constructs for formal specification of OO systems can be found in Object-Z. We use Object-Z to write formal specifications of our core concern. Figure 4 reflect a state schema for the S*tack* class in Object-Z. Stack class has number of methods which are specified in the given schema, figure 2. Hence we complete first step to the formal verification of AO program i.e., the specification of the core concern which is stack implementation is this. Stack schema uses the constructs of Object-Z to write formal specification of the classes. Figure 2 states the specification of the required methods which need to be there in implementation. Object-Z is a very practical solution towards writing classes. In fact the problem lies in specification of the aspects which are never specified before so additionally, we believe that the formal definition of aspects is necessary for analyzing their behavior. It is mandatory to specify the behavior of aspects. The following section 3.2 elaborates the specification of an aspect.

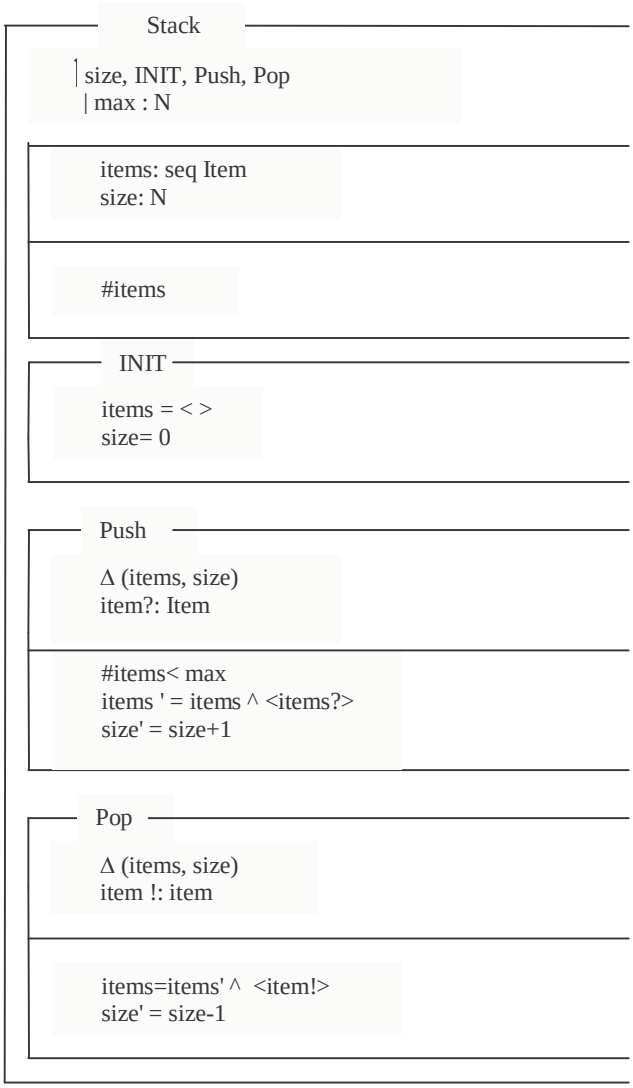

Fig. 2. Formal Specification of Stack class

3.2 Formal Specification of Aspects

Formal specification of stand alone aspect is feasible to understand and adopt. An aspect is formally specified using our own abstract model introduction of some new constructs and notations, whilst there doesn't exist significant work on formal languages for AOP except one, Aspect-Z [19]. However, Aspect-Z doesn't contain

rich notations and constructs. The formal specification of aspect is written by employing our methodology using some ad hoc notations in figure 3. Besides it, we want to ensure that the execution order of aspect with base classes must be preserved. Therefore framework also deals with the requirement of step-wise definition of the constructs, which have been defined in any aspect, are also preserved.

To avoid from unnecessary details the same syntax of Object-Z is maintained. The visibility list in aspect specification contains all the constructs definitions which may be used for some core concerns. While we define the advices as type and their instantiation is possible. Pointcut is an abstract point and we specify it by describing the target for pointcut while the methods which are changed, present their. The definitions of advices are written for specification, autonomously.

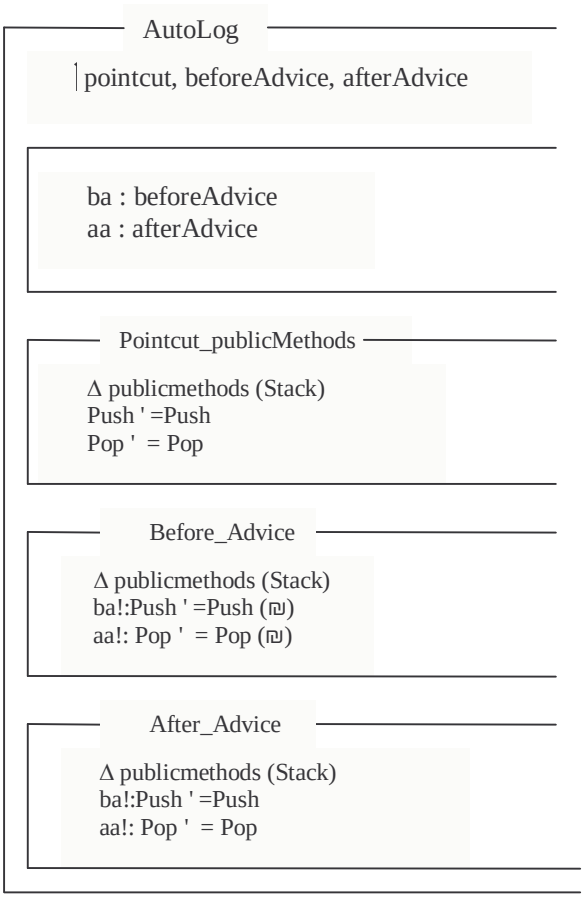

Fig. 3. Formal specification of AutoLog aspect

In the above aspect schema, figure 3, we represent any aspect by relating it to the class schema which affect from it as this is defined in pointcut public methods. In our example, we have one class (Stack) and a aspect (AutoLog), and there exist a pointcut

which is mapped to the public methods of the classes. A pointcut relates to different join points, so we define them in the context of pointcut with an operator (≋) called join point operator. Another major construct is advices; we treat them as relating to some pointcut in the base class so all the advices are specified one by one. The constructs used here are the same as Object-Z, further syntax and constructs can be found in [16].

3.3 AOP - Filtered Specifications

Static definition of classes helps us to define the behavior of a core concern. Object-Z is a standard formal language to specify OO systems, whilst it lacks some constructs or notations to fulfill our requirements for AOP. Its failure scenarios; for example if we need to specify time constraints for real time systems, Object-Z doesn't allow. Along with this, it is also impossible to specify concurrent application using Z-language or Object-Z. RTOZ [28] is one of the extension to the Object-Z language which inherits its class schema style and introduces filter specifications for Real-Time systems. Same trouble goes with AO programs where there doesn't exist any acceptable solution for AO programs specification.

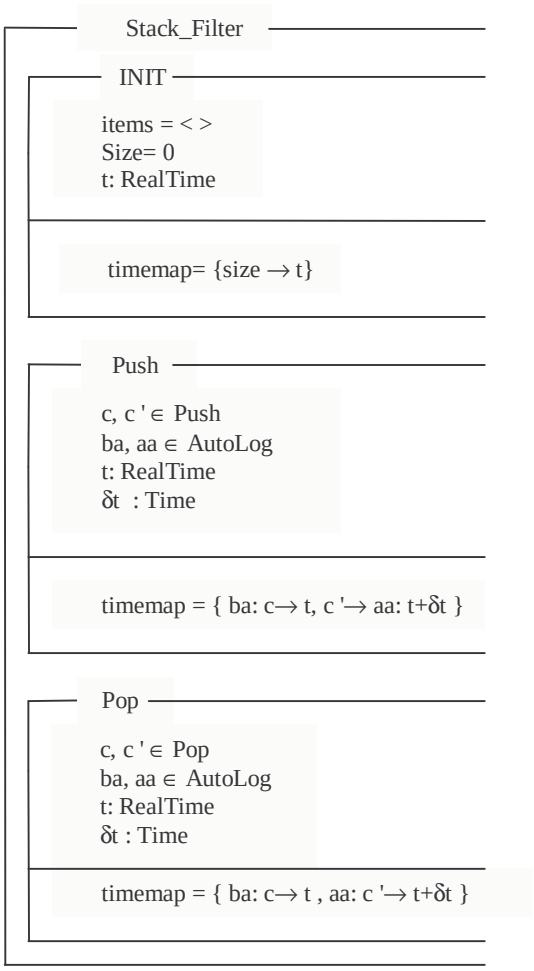

Fig. 4. Filtered specification approach to AO programs.

Hence we use the notion of filtered specification for AOP. C & C′ are pre and postconditions consecutively. The two constructs, before and after advice are introduced to show their relation with the base class while crucial-ness lies in the weaving of advices with the pre and postconditions. Through filtration we preserve the execution sequence and also map the advices to the methods. The constructs are easy to comprehend and practice. We are further investigating filtered specifications to use, which has a most significant advantage in the form minimum constructs' addition to existing Object-Z. Formalists don't need to go under an intensive practice but just some new notations along with matured Object-Z schema works. We introduce preconditions and postconditions format to weave formal specifications Figure. 4. presents the approach in pre and post conditions which reflects a viable solution, however, there are some issues which still need to be resolved during static weaving, specially the problem of around advice. All the methods which reside in Stack class schema are enhanced with the advices of an aspect in a filtered model. Then we use the INIT construct to initialize the relevant classes and aspects. Methods lie in form of pre- and post-conditions while preserving their own pre- and post-conditions. We can also write separate state schema for each method in the class, but we don't follow this methodology in our proposed approach

3.4 Generalizing Formal Specification Constructs

The given example covers only one scenario where we have a single class and an aspect. In real-world systems, this would not be the case. In this section, we are generalizing our approach by providing some basic formal notations for the constructs like pointcut, join point, and advice.

A logical representation of our system would be defined as:

- Advice: Pointcut X JoinPoint
- Let 'PC' be a pointcut, then Q would be the set of pointcuts defined in an aspect:
  [Pointcut]
  [JoinPoint]
  PC: Pointcut
  Q={PC1, PC2, ….PCn}
  Q: P P
- PC: P Join Point
- A pointcut 'PC' can have one or more joint points associated with it, and is defined as:
  PC={JP1, JP2,….,JPn} or we may have
- PC={JPi} where i=1 to n, Hence we can formulate an expression for advices as follows:
  ∑ (I to n) JP € PCj | there exists adv k where j=1 to n and k={before, around, after}
- For integration of the specifications, Before/ After advices with the class schema's preconditions/ postconditions are written as follows:

- timemap = { ba: c→ t , aa: c '→ t+δt }, where using the filters notion the advices are specified to maintain their execution sequence.
- The join point operator (₪) is used to specify join points as these are known as principled or specialized points

One or more aspects are weaved to a class where we need to cater multiple postconditions/ preconditions, there may occur a condition where the contradiction among the advices is possible, especially in case of around advice, as previously discussed. We are currently working on its compositional complexity.

```
public class Stack {
     private int size;
     public Stack ( ) {
     size=0; }
       public void additem
       (Item item) {
     //unimplemented condition
     items.add(item);
     }
       public void removeItem
        ( Item item) {
     //unimplemented condition
     items.remove(item);
     }
     public Boolean empty ( ) {
     return  size = = 0;
     }
     public Boolean full ( ) {
            return size==maxSize;
     }
}
```

Fig. 5. Implemented Stack Class.

```
public aspect AutoLog {
     pointcut publicMethods ( );
       execution      (public      *
     *..*(Item)) && target(Stack);
       Before ( ): publicMethods ( )
     {
       System.out.println("Enter    "
     +thisJoinPoint  .getSignature  (
     ).toString ( ) ); }
       after ( ) ; publicMethods ( )
     {
       System.out.println ("Exit " +
       thisJoinPoint.getSignature ( )
     .toString ( ) );
     }
```

Fig. 6. The AutoLog Aspect .

Class reveals *Stack* operations while in the aspect code; a pointcut is defined along with advices (before & after). Aspect code is weaved into *Stack* class. The implementations given in Figure 5, 6 are based on formal specification of AOP and we may use their compiled code (by a weaver) for reverse engineering to accomplish a comprehensive case study for this framework.

3.5 Formal Specification of the Augmented System

We are always more concerned with the aspectual behavior. In case of our approach, as a second step, we adopt a de-compilation approach for writing formal specification of the compiled code, to accomplish our framework methodology. We need to check that whether the system's behavior is relating to the initial formal specifications, and if not, then what sorts of changes are made to the implemented

core concerns. Through formal specification of the de-compiled classes, we can verify our primary formal definition of the system. We have following verification purposes; usually a verification technique must tackle: these are specific to the core concerns, for instance, types of changes made to the use of the variables, whether the data members of any class are consistent what it was primarily formally specified and specifically the methods are called with the right member and proper types of arguments. Along with this we need to see that initially type descriptors are consistent with the preliminary definitions and constructs and classes with respect to methods, variables, constants etc., exist in weaved code or not.

The aforementioned issues are due to the invasive aspect changes made to the system, our approach also handles them. Aspect specific faults [21] i.e., incorrect strength in pointcut patterns like, incorrect aspect precedence, failure to establish expected postconditions, failure to preserve state invariants, incorrect focus of control flow, incorrect changes in control dependencies can also be determined by the use of our formal verification mechanism as we may get all the possibilities of aspects while weaving to the classes. Adding to these we may have some other faults i.e., faults in control flow changes, inter-type declarations, polymorphic calls, specific issues to join points, any undesirable feature interaction between object, and the aspect, so there may arise situations where the behavior is completely unobservable, what we observe. Even one of the major concerns should be to face the challenge of incompatibilities among aspects and classes. Our framework enables to check these inconsistencies from classes and aspects which we have for weaving. Our proposed formal verifications facilitates to find out maximal faults detection residing in an AO program. Firstly formally specified integrated model of classes and aspects eventually becomes a verification criteria, therefore the weaved system satisfies it or may become a source for us to detect changes which are indeed helpful to evaluate the aspects against the expected behavior.

4. Possible Automation

Figure. 7 illustrate the procedure for automation of our framework. As a fundamental part of our framework, a tool for automation can complement it and possibly more robust to analyze of AO software. We are currently working on a prototypic tool for the initial evaluation of our approach, and will present it in later versions of this paper. Fig. 7 illustrates the block diagram and main working of the tool for the proposed framework. The java byte code is decompiled and then HOL representation is generated using a transformation parser. The compiled aspects and classes specifications are also transformed to HOL format. A verifier then compares both representation for dissimilar constructs and ambiguities, if any. Commonalities reflect that aspects are ensuring required behavior while differences illustrates that some of the required behavior couldn't be achieved, are extra, or is changed. Hence we may able to find out the problems in either aspects or classes. Theorem provers [22], [23] provide support for algebraic proofs of model properties. Examples include ACL2, Alloy, eCHECK (Prover Technologies), KIV, PVS (SRI Inc.), TRIO-Matic, VSE II.

Feil! Objekter kan ikke lages ved å redigere feltkoder.

5. Conclusion & Future Directions

The paper presented a novel approach to formally specify and verify AO programs. First, we formally specify base classes and aspects individually. After this, the integrated model of formal specification (class & aspect) is achieved. The byte code is then reverse engineered into intermediate representation to formulate new classes. Ultimately formal specifications for these classes are written. In the end both the specifications i.e., static definition and formal specifications of intermediate representation are analyzed against each other. A possible tool support illustrates the ease of automation, as we may use theorem proving environment. Our approach also enables us to fo by the classes against the serv methods with rich syntax and constructs are required to deal with complex behaviors of AO systems.

Fig. 7. Tool Architecture for Proposed Framework

The novelty lies in the framework as it is using a best possible combination of widely known Object-Z language, filtered specifications notion, writing formal specifications from the requirements and finally comparison methodology along with the possible automation. Relatively the given approach provides a generic solution as far as the AO software is concerned in the form of whole system specifications.

In our future work, we are focusing to present thorough approach with a case study and prototype tool support.. We are eventually more concerned with internal data structure of the classes, in what circumstances that is endangered, and how it can be analyzed effectively. In addition to this, we are working on a formal method with rich constructs, to address the problem of formal specifications of AO programs.

References

1. Sereni, D., and Moor, O. D., Static Analysis of Aspects, Oxford. In AOSD 2003, Boston, MA USA, in the proceedings of ACM, 2003, Pages: 30 – 39, ISBN:1-58113-660-9
2. Denaro, G., and Monga, M., An Experience on Verification of Aspect Properties, IWPSE 2001, Austria, in the proceedings of ACM, 2001, Pages: 186 – 189, ISBN:1-58113-508-4
3. Ubayashi, N., and Tamai, T., Aspect-Oriented Programming with Model Checking, AOSD 2002, Enschede, The Netherlands, in the proceedings of ACM, 2002, Pages: 148 - 154 , ISBN:1-58113-469-X
4. Katz, S., A Survey of Verification and Static Analysis for Aspects, AOSD-Europe Technion-1(10 July 2005).
5. Larsson, D., and Alexandersson, R., Formal Verification of Fault Tolerance Aspects, in the proceedings of ISSRE, 2005.
6. Mousavi, M., Russello, G., Chaudron, M., Reniers, M.A., Basten, T., Corsaro, A., Shukla, S., Gupta, R., and Douglas C. Schmidt, Aspects + GAMMA = AspectGAMMA, in the proceedings of FOAL, 2002.

7. Hailpern, B., and Santhanam, P., Software debugging, testing, and verification, in the proceedings of IBM SYSTEMS JOURNAL, VOL 41, NO 1, 2002.
8. Weston, N., Taiani, F., and Rashid, A., Modular Aspect Verification for Safer Aspect-Based Evolution, in the proceedings of RAM-SE'05.
9. Krishnamurthi, S., Fisler, K., and Greenberg, M., Verifying Aspect Advice Modularity, in the proceedings of SIGSOFT'04/FSE-12, ACM, 2004.
10. Douence, R., Fradet, P., and Südholt M., Composition, Reuse and Interaction Analysis of Stateful Aspects, in AOSD 04 (March 2004), Lancaster UK, in the proceedings of ACM, 2004.
11. Walker, R.J., Baniassad, E.L.A., and Murphy G.C., An Initial Assessment of Aspect-oriented Programming, ICSE '99 Los Angelos CA USA, in the proceedings of ACM, IEEE, 1999.
12. Sakurai1, K., Masuhara, H., Ubayashi N., Matsuura1 S., and Komiya1 S., Association Aspects, AOSD (March 2004), in the proceedings of ACM, 2004, Pages: 16 – 25, ISBN:1-58113-842-3
13. Torzer, M., Krinke, J., and Breu S., Trace Analysis for Aspect Application, in the proceedings of AAOS, 2003.
14. Okum, V., Black, P.E., Issues in Software Testing with Model Checkers, in the proceedings of DSN, 2003.
15. Rinard, R., Salcianu, A., and Bugrara S., A Classification System and Analysis for Aspect Oriented Programs, in the proceedings of ACM, Pages: 147 – 158, ISSN:0163-5948, 2004.
16. Smith, G., The Object Z Specification Language, Kluwer Academic Publishers, 2000, ISBN: 0792386841.
17. AspectC++ Homepage: http://www.aspectc.org.
18. AspectJ Homepage: http://eclipse.org/aspectj.
19. Yu, H., Liu, D., Yang, L., and He, X., Formal Aspect-Oriented Modeling and Analysis by AspectZ*, in the proceedings of the SEKE, 2005.
20. Xu, W., Xu, D., Goel, V., and Nygard, K., Aspect flow graph for testing aspect-oriented , Department of Computer Science, North Dakota State University Fargo, ND 58105. U.S.A.
21. Alexander, R.T., Bieman, J.M., and Andrews A. A., Towards the Systematic Testing of Aspect-Oriented Programs, Technical Report, Colorado State University, Department of Computer Science, 2003.
22. James R., and Slagle., Automated Theorem-Proving for Theories with Simplifiers Commutativity, and Associativity, Journal of the ACM (JACM) archive, volume 21 , Issue 4, pages: 622 – 642, 1974.
23. Winterstein D., Bundy A., and Gurr C., Dr.Doodle: A Diagrammatic Theorem Prover, in the proceedings of the IJCAR 2004.
24. Achim D. Brucker., Rittinger F., and Wolff B., HOL-Z 2.0: A Proof Environment for Z-Specifications. In Journal of Universal Computer Science, 9 (2), pages 152-172, 2003.
25. John K., and Dixon., Z-Resolution: Theorem-Proving with Compiled Axioms, Journal of the ACM (JACM), Volume 20 , Issue 1, Pages: 127 – 147, 1973, ISSN:0004-5411.
26. Proebsting T.A., and Watterson S.A., Krakatoa: Decompilation in Java (Does Bytecode Reveal Source?), Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems, Portland, Oregon, June 1997.
27. Naeem N.A., and Hendren L., "Programmer-friendly Decompiled Java," icpc, pp. 327-336, 14th IEEE International Conference on Program Comprehension (ICPC'06), 2006
28. K. Periyasamy, and V.S. Alagar, "Adding Real-Time Filters to Object-Oriented Specification of Time Critical Systems," wift, p. 28, Second IEEE Workshop on Industrial Strength Formal Specification Techniques, 1998.
29. Gunter, Elsa L., Felty, and Amy., Theorem Proving in Higher Order Logics, 10th International Conference, TPHOLs'97, Murray Hill, NJ, USA, August 19-22, 1997,

Proceedings, Series: Lecture Notes in Computer Science , Vol. 1275,1997, VIII, 339 p., Softcover, ISBN: 3-540-63379-0
30. Jones R.B., Z in HOL - the story of ProofPower, 13 April 2006
31. W. Lim., J. Harrison, P. Bailes, A. Berglas, "Design Recovery through Formal Specification," aswec, p. 22, Australian Software Engineering Conference, 1998.
32. Antoy S., Hamlet D., "Automatically Checking an Implementation against Its Formal Specification," IEEE Transactions on Software Engineering, vol. 26, no. 1, pp. 55-69, Jan., 2000.