Software Engineering 2013, 3(2): 5-12
DOI: 10.5923/j.se.20130302.01

Change Operations and Their Consequences in AOP Evolution

Sandra Casas*, Cecilia Fuentes Zamorano, Héctor Reinaga

Instituto de Tecnología Aplicada, Universidad Nacional de la Patagonia Austral, Río Gallegos, 9400, Argentina

Abstract Simple change operations to source code can cause unexpected behavior. In particular, two well-known problematic situations, fragile pointcuts and aspect interactions, pose serious challenges for the evolution of software with aspects. We believe that if developers are informed about the consequences of the change operations they are considering, they will be able to avoid various errors. This work analyzes the effects of change operations in AO applications. It proposes the Identification, Qualitative, and Quantity (IQQ) model as a conceptual approach to anticipating the consequences of change operations, along with BaLaLu, a tool that supports the IQQ model.

Keywords Aspect-oriented Programming, Separation of Concern, Change Operations, Software Evolution, AspectJ

1. Introduction

A crosscutting concern (CCC) is program behavior that cannot be adequately modularized with respect to the other parts of a system[1]. AOP provides constructs for modularizing CCCs[2] in order to decrease code scattering and tangling. AOP proposes a new kind of modularization called aspects. An aspect is a module that can localize the implementation of a CCC. AOP adopts a specific conception of CCCs: a CCC contains functionality that is executed at different join points. A join point is a well-defined point in a program's control flow. The main abstractions of AOP are pointcuts, which are predicates that describe a set of join points, and advice, comprised of blocks of functionality that can be bound to pointcuts. The key to the AOP modularization technique lies in its composition mechanism. In traditional approaches such as OO, subroutines explicitly invoke the behaviors implemented by other subroutines. In contrast, aspects have an implicit invocation mechanism, so that the behavior of an aspect is implicitly invoked in the implementation of other modules. Consequently, the implementation of these other modules can be largely unaware of the CCC.

However, this structure (pointcuts and advice) makes it difficult for developers to evaluate the behavior of a system. In particular, the implicit invocation mechanism introduces an additional layer of complexity in the construction of a system. This can make it difficult to understand how and when the base system and the aspects interact, and consequently, how the overall system will behave.

Moreover, seemingly innocuous and simple changes in the source code may produce erroneous and unintended behaviors. Since it is easy to lose track of the global characteristics of how base code and aspects interact, it can be difficult to identify the code that is responsible for such unanticipated behavior. Similar consequences can occur when a join point is reached by two or more pointcuts and the developer is not aware of the situation and therefore does not address it. These two problematic situations, known as *fragile pointcuts*[3] and *aspect interactions*[4], represent a real problem for the evolution of software using aspects and can directly impact maintenance tasks, time, effort, and costs.

Our proposal aims to answer the question, What will happen if a change operation is performed? We believe that if developers are aware of the consequences of future change operations, they can avoid various errors. The central software artifact in this study is source code, and for that reason we have focused on the AspectJ language[5]. However, our approach may be applied to any other AO language.

The rest of this paper is organized as follows. In Section 2, we analyze the effects of change operations in AO applications. In Section 3, we present the Identification, Qualitative, and Quantity (IQQ) model as a conceptual approach to anticipating the consequences of change operations. In Section 4, we present the BaLaLu tool, which supports the IQQ model, along with some tests. Finally, in Sections 5 and 6, we discuss related work and present our conclusions.

2. AOP Evolution

* Corresponding author:
scasas@unpa.edu.ar (Sandra Casas)
Published online at http://journal.sapub.org/se

Aspects are the central abstraction of AOP. Two modular components comprise an aspect: pointcuts and advice. Advice is a fragment of code, such as a method, that will be executed with the base code (after, before, or around). A pointcut is an expression that establishes the events and conditions specifying when and where advice code will be executed, typically as a method call. Pointcuts are the more critical elements in AOP evolution, because a simple change in the base code may alter the set of join points of any pointcut and have consequences for advice execution. Pointcuts can refer to events either explicitly or by using defined pattern names with wildcards. The tight coupling and dependence between pointcuts and base code are the cause of fragile pointcuts and aspect interactions.

Change operations represent software evolution; they are the actions that developers carry out when they modify source code. Some examples include adding a class, renaming a method, and applying a refactoring[6]. Change operations are important because they can generate diverse nonlocal consequences in AO applications. After a change operation has been applied, a pointcut may either capture too many join points (false positives) or fail to capture certain join points that were intended to be captured (false negatives). A simple example is presented in the following listing code:

```
public aspect LogChangePosition {
 pointcut changePositionPoint(Point p):
    call(* Point.set*(int))&& target(p);
 after(Point p): changePositionPoint(p) {
      Logger.writeLog("Change position Point:"
      +p.toString());  }
 pointcut changePositionLine(Line l):
      call(* Line.set*(Point)) && target(l);
 after(Line l): changePositionLine(l) {
  Logger.writeLog("Change position Line:"
      +l.toString());  }
}
```

The LogChangePosition aspect implements a log mechanism to register the position changes of Point and Line objects. Table 1 indicates consequences that very simple changes in the domain (Point and Line classes) can have.

Table 1. Change Operations and Consequeces

Change	Consequences
Rename type Point as MyPoint	The interception {call (void Point.set*(int)} is empty and potential false negatives result.
Change the signature void setX(int) to void setX(double)	{call (void Point.set*(int)} is broken and potential false negatives result.
Add a field to Point and Line classes that is not related to position, and add a setting method for it.	LogChangePostition aspect will intercept the calls to the new method and potential false positives result.

The problems depicted are known as "fragile pointcuts"[3] and are a real problem for the evolution of software with aspects. Related issues are discussed in[7, 8, 9, 10].

Interaction, conflicts or interferences[4] comprise another issue that may arise during software evolution in complex systems with several CCCs and aspects. A new pointcut can create one or more conflicts with existing pointcuts. Sometimes conflicts among aspects require specific treatment by the developer, such as defining an order of execution. The problem arises because weavers of the AO language do not report when aspects are in conflict, and they are simply weaved (composed) as in any other case. If an application developer is not aware of the conflicts among aspects, the application behavior may be erratic and unpredictable.

An elementary change in base code or a pointcut specification can produce potential false positives/negatives and interactions. When this happens, developers must identify the problem and resolve it. However, the identification of false positives/negatives and interactions and their causes is not a trivial task in medium-scale applications. This analysis is even more difficult when it is performed after the source code has been modified, at which point developers must perform several tasks such as exhaustive code analysis and inspection and intensive execution of test cases. All these tasks impact maintenance time and effort, with increasing maintenance costs, and new methods and tools are necessary to reduce the maintenance time, effort, and costs.

3. IQQ Model

The goal of the IQQ model is to provide a model for anticipating the consequences of changes in applications with aspects. IQQ does this based on three premises:

1) Identify the consequences of change operations in AO applications. This implies the possibility of detecting the effects of change operations on source code.

2) Quantify the consequences of change operations in metrics that facilitate the analysis for developers. This implies the possibility of quantitatively measuring false positives/negatives and conflicts that a change operation may produce.

3) Qualify the consequences and relate them to the quantified information. This implies the possibility of delimiting the segments of source code that may be affected by a change operation.

The main components of the IQQ model are a program repository, change operations, and their consequences.

3.1. Program Repository

Our approach represents programs as entities rather than text files. Since we focus on AO applications, we consider constructs such as packages, classes, methods, fields, aspects, pointcuts, advice, and exception handlers. We also represent different relationships among these entities that are relevant for AO, such as inheritance, method calls, and aspects weaving/compositions. Each entity has several properties

and states such as identifier, type, and access modifier. These properties and states identify and represent entities in the repository and the relationships between them. Pointcuts are represented in two ways, as expressions and as sets of join points intercepted in specific instances.

Figure 1 represents the main entities and relationships in the repository.

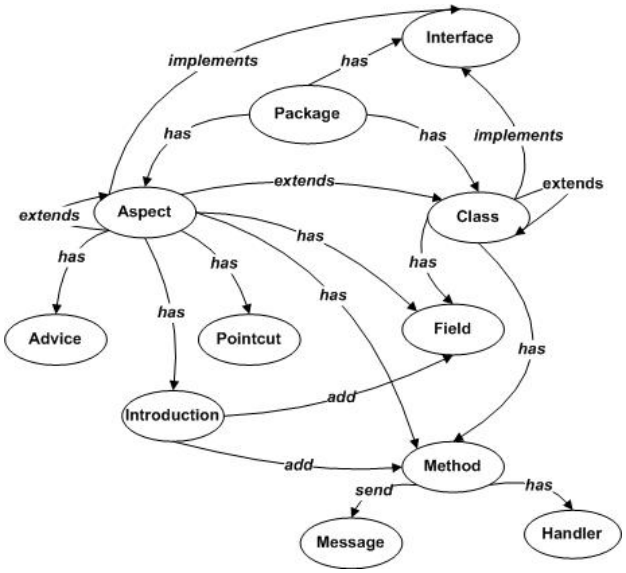

Figure 1. Entities and relationships in the repository

3.2. Change Operations

In the IQQ model, a change operation is a function whose inputs produce specific outputs over a specific instance of the repository. The IQQ model considers both atomic and composite change operations.

Atomic Change Operations: These are indivisible operations that cannot be separated into more than one task or step; thus, they are very simple. An atomic change operation contains all the necessary information to represent a function that can be analyzed with the repository information. An atomic change operation can produce false positives/negatives as well as conflicts during system evolution. The following atomic change operations suffice for the IQQ model:

- Add a package/class/method/field/handler/message
- Remove a package/class/method/field/handler/message
- Add/remove a pointcut
- Add/remove a declare parents
- Add/remove an advice.

Composite Change Operations: A composite change operation is a sequence of atomic change operations. Sometimes the order in which the atomic operations should be done is mandatory. For instance, the operation "move a class" may be split into "remove a class" and "add a class" atomic change operations. The set of false negatives / positives that a composite change operation can produce is the union of the individual results of each component atomic change operation. The following composite change operations suffice for the IQQ model:

- Move a class/method/field/handler/message
- Rename a package/class/method/field
- Rename a pointcut
- Change a declare parents
- Change a pointcut
- Change an advice.

The change operation "change a pointcut" includes several actions such as changing a primitive pointcut designator (for example, from "call" to "execution") or changing a join point expression (for example, from "Account.debit(..)" to "Account.*(int))".

In [11], we present a complete specification of atomic and composite change operations.

3.3. Consequences

In previous Section we show a basic example of how simple change operations can generate potential false positives/negatives. In[12], we analyzed in depth the potential consequences of each change operation over pointcut expressions of AspectJ. Usually, the "add" change operations can generate potential false positives; the "remove" change operations can generate potential false negatives; the "move" and "rename" change operations can generate potential false positives/negatives; and "change a pointcut" can generate interactions and/or false positives /negatives. For example, the change operation "remove class X" impacts all designators of pointcuts that refer to class X. That is, the join point expressions of a primitive pointcut designator include "call", "execution", "target", "within", and so on, and if X is referenced in any of these expressions, then a potential false negative is present.

In general, we say

if (ChOp(x) && P(x)) then[C],

where ChOp is any change operation, P is any pointcut of the application, x is a source code entity (package, class, method, field, pointcut, advice, etc.), and C is the set of consequences of ChOp (false positives/negatives and interactions).

4. BaLaLu

BaLaLu is a tool that we have developed to support the IQQ model. BaLaLu can analyze 31 change operations, 18 of which are atomic operations and 13 of which are composite change operations. Among these operations, 21 are change operations over base code and 10 are change operations over aspects code. BaLaLu supports both Java and AspectJ source code.

4.1. Design and Implementation

The change operations comprise a hierarchy in which the atomic and composite classifications are the main subclasses. AddClass, AddPackage, AddMethod, AddField, AddAdvice, AddMessage, RemoveClass, RemoveMethod, RemoveField, RemoveMessage, RemoveAdvice, and so on are subclasses

of AtomicChange. MoveClass, MoveMethod, MoveField, MoveMessage, RenameClass, RenameMethod, RenameFiel d, and so on are subclasses of CompositeChange.

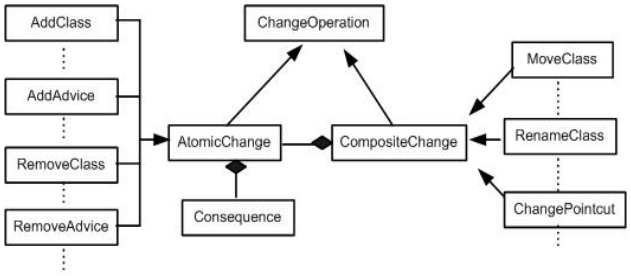

Figure 2. Diagram of classes of change operations

Each atomic change has consequences (potential false positives/negatives and/or interactions). Composite change objects are composed of sets of atomic change objects. Consequence objects represent information about false positives/negatives or interactions (such as join points, aspects, or pointcuts) that will be given to users. Figure 2 is a simple schema of the design.

The repository manages the entity-relationships model that represents program source code. The repository is implemented as a relational database. Each change operation class has a specific SQL query to execute. The parameters of the query are fields of the particular change operation class.

Figure 3 presents a very simple scenario in which we need to evaluate the consequences of removing the setBalance method of the Account class.

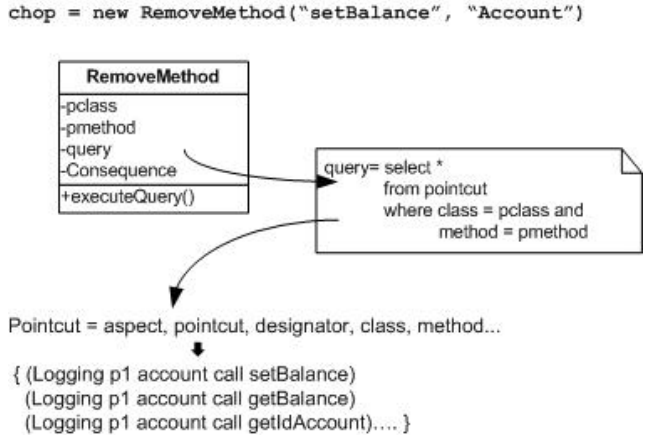

Figure 3. Instance of atomic change operation

The Logging aspect is matching calls of all methods of the Account class. The Pointcut table contains all join points matched by each pointcut. An instance of the RemoveMethod class is created with "setBalance" and "Account" fields. Then a query is set up with these values. The executeQuery method executes the query and maps the results to Consequence objects.

A general template (form) with optional fields is used to select change operations and configuration input parameters. This makes it easy for a developer to define a change operation. BaLaLu shows the results with a report. The numbers of false positives/negatives and interactions are shown in the upper part of the report panel, and a detailed description of the source code elements (package, class, method/field, aspect, pointcut, etc.) affected by the change operation is shown in the lower part of the report.

The developer can define filters, which can be applied over different entities (aspect, pointcut, or class), to limit enclose the list of results.

4.2. SPACEWAR Example

Spacewar is an implementation of the classic video game. The source code is distributed by Eclipse. Spacewar has 3053 lines of code, including 2 packages, 17 classes, 8 aspects, 127 methods, and 21 pointcuts.

We have used BaLaLu to analyze several Spacewar change operations. First, we specify a set of new requirements, in order to determine the necessary change operations to implement them. For example, the requirement "count the firings by game" requires two change operations:

a) add a new pointcut to intercept all calls of the fire method of the "Ship" class

b) add new advice associated with the pointcut.

Next, we enter these change operations in BaLaLu, to probe the following change operations:

ChOp#1.Remove "register" method from "Registry" class

ChOp#2. Remove "Player" class from "Spacewar" package

ChOp#3. Add "Boat" class to "Spacewar" package.

ChOp#4. Add "getActivate" method to "Ship" class

ChOp#5. Add "setSuccess" method to "Ship" class.

ChOp#6. Rename "Spacewar" package by "Armageddon"

ChOp#7. Move "bounce" message to Ship class.

ChOp#8. Move "Registry" class to "Coordinator" package

ChOp#9. Rename "Ship" class as "Boat"

ChOp#10. Rename "clockTick" method of "Game" class by "seconds"

ChOp#11. Move "newShip" method to "Registry" class

ChOp#12. Add pointcut "fire" with expression "call(void Ship.fire())" to Debug aspect.

ChOp#13. Add pointcut "minimum" with expression "call(boolean Ship.expendEnergy(double amount)) && args(p) && if(p.getEnergy() < 0.10)" to Debug aspect

ChOp#14. Change pointcut (join point) "call(Ship Game.newShip(Pilot)) && args(p)" of SpaceObjectPainting aspect to " call(Game.*(..))".

ChOp#15.Remove pointcut "call(Game+.new(String)) from DisplayAspect aspect.

ChOp#16. Change pointcut (designator) of Debug aspect to "execution(* (spacewar.* && !(Debug+ || InfoWin+)).*(..))" .

ChOp#17. Change pointcut (designator) of Debug aspect to "execution(void Ship.bounce(Ship, Ship)) && args(s, s1)".

ChOp#18. Change pointcut (designator) of Debug aspect to "target(r) && (call(void register(..)) || call(void unregister(..)))".

ChOp#19. Add pointcut "call(Robot.*())" to Debug aspect.

ChOp#20. Add pointcut "preinitialization((spacewar.* && !(Debug+ || InfoWin+)).new(..))" to Debug aspect.

ChOp#21. Remove pointcut "call(Game+.new(String)) from DisplayAspect aspect.

Figure 4 presents the consequences calculated by BaLaLu. In the graph, we also contrast the quantity of join points affected by these change operations with the current join points matched by the original pointcuts (violet bar).

All change operations have potential consequences, because the Debug aspect matches all join points (method and constructor calls) of the Spaceware package.

The Spaceware package contains most of the functionality of the application, so that any change causes potential consequences. As we said earlier, in general, "add" change operations can cause potential false positives (ChOp#3, ChOp#4, ChOp#5, ChOp#12, ChOp#13, ChOp#19, and ChOp#20); "remove" change operations can cause potential false negatives (ChOp#1, ChOp#2, ChOp#15, and ChOp#21); and composite change operations can cause all types of consequences (ChOp#6, ChOp#7, ChOp#8, ChOp#9, ChOp#10, ChOp#11, ChOp#14, ChOp#15, ChOp#16, ChOp#17, and ChOp#18).

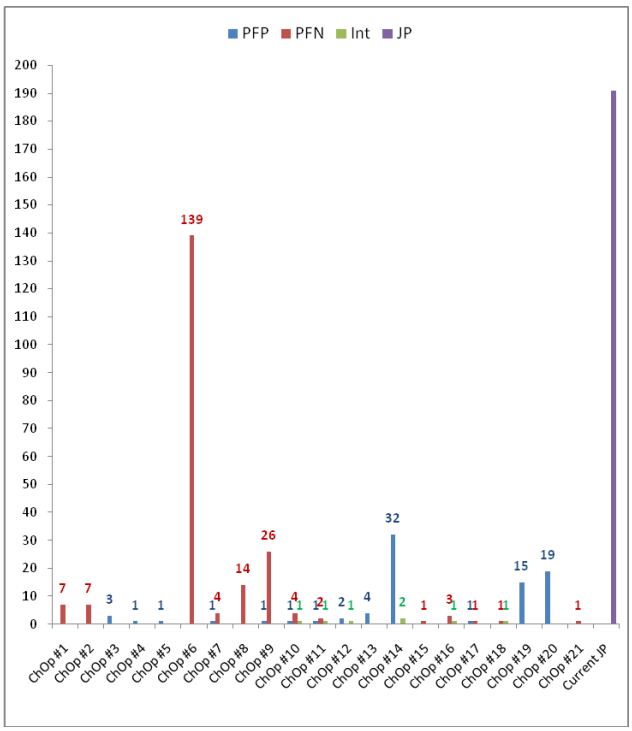

Figure 4. Consequences of change operations in Spaceware

4. Related Works

SpyWare[13, 14, 15, 16] and EclipseEye[17] are IDEs that embody the change-based software evolution (CBSE) approach. CBSE arose in opposition to the typical configuration version systems to overcome their difficulties. CBSE treats changes as first-class entities. One difference between the CBSE model and our proposal lies in the purpose of the CBSE model, which defines the history of a program as the sequence of changes that the program has undergone. Based on the history of changes, a developer can reconstruct each successive state of a program's source code. In addition to this difference, which we consider substantial, CBSE treats change operations as first-class entities while the IQQ model defines them as functions, and the success of the CBSE model requires that it be implemented in these IDEs or development tools, while the IQQ model can be incorporated into these IDEs or in other specific tools such as BaLaLu. Finally, CBSE applies only to OO applications (Java and Squeak) and does not consider AO applications, although we assume that it is possible to extend the CBSE model to AOP.

Automated tools as AJDT[18] and PointcutDoctor[19] in the face of pointcut expression show the effectively intercepted join points and also the "almost" intercepted ones, which is useful when a change operation occurs in the aspects but is insufficient for change operations that occur in the domain.

AspectMaps[20] is a tool that uses software visualization to aid in the understanding of AO software systems. It provides a scalable visualization of implicit invocations, selected join point shadows, and, if multiple aspects are to execute, the order in which they are specified to run. Another tool that uses software visualization is ITDVisualizer[21], an analysis toolkit for assessing how static and structural declarations impact the method lookup of the base program and for identifying how inter-type declarations shadow particular base code entities. The main differences between these tools and our approach are: a) BaLaLu outputs are textual reports rather than graphical; b) BaLaLu analyzes dynamic CCCs (pointcuts and advice) but not static CCCs (inter-type); and c) while AspectMaps and ITDVisualizer can provide visualizations of the current state of source code, they do not provide visualizations of the effects of future changes.

Several tools such as PCDiff[3], Celadon[22], and Souyoul[23] and an unnamed tool in[24] have been proposed to analyze change impacts for AO programs. In general, these tools analyze and compare two or more versions of source code programs. The observed differences are used to derive a group of atomic change operations. These tools work with abstract representations of programs such as syntax trees, call graphs, and dependence graphs, and they also include test cases. An important difference between these tools and BaLaLu is that BaLaLu is not a tool for analyzing the impacts of changes. But we can also summarize the other main differences between these tools and BaLaLu:

a) These tools outline methods based entirely on comparing program versions; thus, they detect and analyze the impact of changes "after" the changes occur. BaLaLu aims to identify the consequences of changes "before" they occur.

b) Because these tools work with program versions, they only can find differences in terms of "atomic" change operations. BaLaLu can also analyze composite change operations. When a composite change operation is analyzed as a set of independent and dissociated atomic change operations, the results lose semantics and integrity.

c) The analysis and assessment of source code program

versions arose from using CVN or Subversion systems for software evolution and maintenance. The limitations and shortcomings of these tools for improving evolution and maintenance tasks are clearly identified in[25].

Vidock[26] is a tool for analyzing the impact of aspect weaving in test cases. It performs a static analysis that identifies the subset of test cases that are impacted by the aspect weaving. This tool works after the changes in source code have been made. However, Vidock is complementary to our proposal since it can corroborate the results calculated by BaLaLu.

A method to analyze the change impacts of woven aspects is proposed in[27], but the method is not supported by a tool. This work analyzes how aspects can change the control flow, input/output parameters, values of data members, and inheritance dependencies of the base code. It also describes the influences and possible effects of pointcut declarations on inheritance and overriding dependencies and how the ripple effects can be computed.

Table 2 summarizes all of these tools, comparing a) the source code programming language(s) that the tool covers; b) the main objectives of the tool; c) the main approach, technique, or strategy used by the tool; d) the information or results produced by the tool; and e) how the tool is implemented. This table of analysis tools for the maintenance and evolution of software with aspects is not complete, but to date we know of no other tools that analyze software in advance of its implementation or that include composite change operations for AO software.

Table 2. Comparison of Tools

Tool	Language (s)	Objective	Approach – Strategy	What is revealed	Imple-mentation
PCDiff[3]	AspectJ	Change impact analysis.	Comparison of program versions. Call graphs. Test cases.	Atomic change operations over classes and aspects. Interferences between aspects.	Eclipse plugin
SpyWare [13-16]	Squeak	Replace software configuration systems such as CVN and Subversion.	CBSE: change operation as first-class entity, program as AST where each node has its history. Repository of changes.	Atomic change operations and refactoring.	IDE
EclipseEye[17]	Java				
AJDT [18]	AspectJ	Develop programs. Editing and compilation.	Software visualization.	Pointcut expressions.	Tool suite in Eclipse IDE
PointcutDoctor [19]	AspectJ	Help developer write correct pointcuts.	Heuristic rules, relaxation process. Recursive explanation.	Join points matched and not matched by pointcut.	AJDT plugin
AspectMap[20]	Java AspectJ	Provide aspects understanding. Visualization that shows how aspects crosscut the base code, as well as how they interact at each join point.	Software visualization.	Implicit invocations. Join point shadows. Order of execution of aspect interactions.	AJDT plugin
ITDVisuali-zer [21]	Java AspectJ	Assess the impact of structural modifications made through AspectJ inter-type declarations on the behavior of the system.	Analyzes structural information about a program before and after weaving. Uses a modified AspectBench compiler (abc).	Interaction patterns bet-ween the static crosscutting construct of AspectJ and base programs: lookup impact, shadowing impact, orthogonal.	IDE Eclipse
Celadon[22]	AspectJ	Understand the impact of program changes.	Comparison of program versions. Abstract syntax tree and static call graphs. Dynamic programming algorithm and RTA algorithm.	Atomic changes together with relationships. Subset of regression tests that are impacted by those changes.	Not mentioned
Souyoul[23]	AspectJ	Change impact analysis.	Comparison of AspectJ program versions. Dependency graphs. Program slicing.	Atomic change operations over aspects.	Not mentioned
[24]	AspectJ	Change impact analysis.	Comparison of program versions. Control flow graphs. Analysis of syntactic and semantic differences. Test cases.	Atomic change operations over aspects and classes.	Top abc compiler
Vidock[26]	AspectJ	Calculate test cases impacted by aspects.	Static analysis of program and test cases. Abstract syntax tree. Static call graphs.	Impacted test cases.	IDE Eclipse
Balalu	AspectJ	Anticipate the consequences of atomic and composite change operations over classes and aspects.	Repository of program structures. Change operations are SQL queries.	Potential false positives/negatives. Aspect interactions.	Stand-alone

5. Conclusions

In this work, we have analyzed a repertory of change operations that can have unintended consequences in AO applications during software evolution. The main reasons for these problems are fragile pointcuts and aspect interactions. We have proposed strategies to anticipate these consequences before the source code is changed. The main contributions of this work are: showing how change operations over source code can produce undesired effects (ripple effects); identifying several of these change operations; identifying their consequences, quantifying them, and locating them in source code; and performing these analyses with an automatic process before the change operations are implemented. Although tests and code inspections cannot be eliminated, the IQQ model and BaLaLu tool can be used to reduce the need for these. This decreases the time and effort necessary for detecting the ripple effects produced by change operations in the context of AO applications.

However, open problems still remain. In software applications with aspects, join points may be activated or disabled for the sake of change operations, and perhaps for other purposes. In general, developers change source code with some goal in mind, and the changes may have any meaning. Isolated atomic change operations are less frequent than composite change operations when new requirements need to be implemented. Nevertheless, the identification of composite operations is strongly tied to atomic operations. The set of atomic operations is large, but we can confine them to those that can be affected by an aspect. For example, "remove a local variable" is an atomic change operation, but it is not relevant for AOP because local variables cannot be intercepted by pointcuts. The universe of composite change operations is anticipated to grow as a result of the recent introduction of "refactorings". Refactoring is more complex than "moving" or "renaming" an entity. Our future work will address the possibility of anticipating the consequences of refactoring in software with aspects. It will also investigate passing BaLaLu to IDE-based tools for Eclipse.

ACKNOWLEDGEMENTS

This work was partially supported by the Universidad Nacional de la Patagonia Austral, Santa Cruz, Argentina.

REFERENCES

[1] W. Hürsch and C. Lopes, "Separation of Concerns", Northeastern University Technical Report NU-CCS-95-03, Boston, 1995.

[2] G. Kiczales, G. Lamping, J. Mendhekar, A. Maeda, C. Lopes, C. Loingtier and J. Irwin, "Aspect-oriented Programming", Proceedings of the European Conference on Object-Oriented Programming (ECOOP), LNCS 1241, Springer-Verlag, 1997.

[3] C. Koppen and M. Stoerzer, "Pcdiff: Attacking the Fragile Pointcut Problem", European Interactive Workshop on Aspects in Software, Berlin, Germany, 2004.

[4] S. Casas, J. García Perez-Schofield and C. Marcos, "Conflictos en AspectJ: Restricciones y Soluciones" Revista IEEE América Latina. Vol. 8 – N 3, 2010, pp 280-286.

[5] G. Kiczales, "Tutorial on Aspect-Oriented Programming with AspectJ", FSE , 2000.

[6] M. Fowler, "Refactoring: Improving the Design of Existing Code". Addison-Wesley, 1999.

[7] R. Coelho, A. Rashid, A. Garcia, F.Ferrari, N. Cacho, U. Kulesza, A. Staa and C. Lucena, "Assessing the Impact of Aspects on Exception Owns: An Exploratory Study", European Conference on Object-Oriented Programming (ECOOP), 2008, pp. 207-234.

[8] S. Soares, P. Borba and E. Laureano, "Distribution and Persistence as Aspects", Software: Practice&Experience., Vol. 36 (7), 2006, pp. 711-759.

[9] E. Figueiredo, N., Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, U., A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Filho and F. Dantas, "Evolving Software Product Lines with Aspects: an Empirical Study on Design Stability", ICSE 08: Proceedings of the 30th International Conference on Software Engineering, USA, 2008, pp. 261-270.

[10] A. Kellens, K. Mens, J. Brichau and K. Gybels, "Managing the Evolution of Aspect-Oriented Software with Model-Based Pointcuts", European Conference on Object-Oriented Programming (ECOOP), LNCS N 4067 in LNCS, 2006, pp. 501-525.

[11] S. Casas, H. Reinaga and C. Fuentes Zamorano, "AspectJ bajo la Lupa", IX WIS – XVIII CACIC – ISBN 978-987-1648-34-4, Argentina, 2012, pp. 714-723

[12] S. Casas, H. Reinaga and C. Fuentes Zamorano "Aplicaciones con Aspectos: de Cambios y sus Consecuencias" 13° ASSE – 41° JAIIO - 2012 - ISSN 1850-2792 – Argentina

[13] R. Robbes and M. Lanza, "An Approach to Software Evolution Based on Semantic Change", Proceedings of Fase 2007, 2007, pp. 27-41.

[14] R. Robbes and M. Lanza, "Change-Based Software Evolution", EVOL 2006, 2006, pp. 159- 164.

[15] R. Robbes and M. Lanza, "A Change-Based Approach to Software Evolution", ENTCS, Vol 166, issue 1, 2007, pp. 93-109.

[16] R. Robbes and M. Lanza, "Towards Change-Aware Development Tools". Technical Report at USI, 25 pages, 2007.

[17] Y. Sharon, "Eclipseye — spying on eclipse", Bachelor's thesis, University of Lugano, 2007.

[18] AJDT: AspectJ Development Tools, http://www.eclipse.org/ajdt/

[19] L. Ye and K. De Volder, "Tool support for understanding and diagnosing pointcut expressions", International Conference Aspect-Oriented Software Development, 2008.

[20] J. Fabry, A. Kellens and S. Ducasse "AspectMaps: A Scalable Visualization of Join Point Shadws". AOSD 2010 – France.

[21] D. Zhang, E. Duala-Ekoko and L. Hendren "Impact analysis and visualization toolkit for statict crosscutting in AspectJ", In 17th International Conference on Program Comprehension (ICPC) 2009. Canada.

[22] S. Zhang and J. Zhao, "Change Impact Analysis for Aspect-Oriented Programs". Technical Report SJTU-CSE-TR-07-01, Center for Software Engineering, Shanghai Jiao Tong University, 2007.

[23] I. Bouteraa and N. Bounour, "Towards The Use of Program Slicing In the Change Impact Analysis of Aspect Oriented Programs", ACIT'2011 Proceedings International Arab Conference on Information Technology – Arabia Saudita, 2011.

[24] L. Cavallero & M. Monga, "Unweaving the Impact of Aspect Changes in AspectJ". FOAL 09 – USA.

[25] R. Robbes and M. Lanza, "Versioning systems for evolution research". In Proceedings of IWPSE 2005 (8th International Workshop on Principles of Software Evolution), IEEE Computer Society, 2005, pp 155– 164.

[26] R. Delamare, F. Muñoz, B. Baudry and Y. Le Traon "Vidock: a Tool for Impact Analysis of Asect Weaving on Test Cases". ICTSS'10 Proceedings of the 22nd IFIP WG 6.1 international conference on Testing software and systems - Springer-Verlag Berlin, Heidelberg ©2010. Pp 250-265.

[27] Ch. Liu, S. Chen and W. Jhu, "Change Impact Analisys for Objetc-oriented Programs Evolved to aspect-Oriented programs", SAC 2011. Taiwan.S. M. Metev and V. P. Veiko, Laser Assisted Microtechnology, 2nd ed., R. M. Osgood, Jr., Ed. Berlin, Germany: Springer-Verlag, 1998.