

Available from: [http://doi.acm.org/10.1145/2351676.2351709](http://doi.acm.org/10.1145/2351676.2351709)
Supporting Automated Software Re-Engineering using “Re-Aspects”

Mohamed Almorsy, John Grundy, and Amani S. Ibrahim
Centre for Computing & Engineering Software Systems
Swinburne University of Technology
Melbourne, Australia
[malmorsy,jgrundy, aibrahim]@swin.edu.au

ABSTRACT
System maintenance, including omitting an existing system feature e.g. buggy or vulnerable code, or modifying existing features, e.g. replacing them, is still very challenging. To address this problem we introduce the “re-aspect” (re-engineering aspect), inspired from traditional AOP. A re-aspect captures system modification details including signatures of entities to be updated; actions to apply including remove, modify, replace, or inject new code; and code to apply. Re-aspects locate entities to update, entities that will be impacted by the given update, and finally propagate changes on the system source code. We have applied our re-aspects technique to the security re-engineering problem and evaluated it on a set of open source .NET applications to demonstrate its usefulness.

Categories and Subject Descriptors
D.2.7 [Distribution, Maintenance, and Enhancement]: Restructuring, reverse engineering, and reengineering

General Terms
Algorithms, Design, Languages

Keywords
System Reengineering; Software Evolution; Change Impact Analysis; Re-Aspects

1. INTRODUCTION
Software systems are usually exposed to extensive change and evolution after deployment. These maintenance activities represent up to 80% of the total system cost and effort [1]. This usually requires capturing new features/modifications, locating system entities that must be modified, locating system entities that are impacted by this modification and should be modified, and propagating the required modification to the whole system. Existing system maintenance approaches are mostly limited to change impact analysis [2, 3], where we identify entities that will be impacted by a given system modification to a specific system entity. These efforts assume that entities to modify are known and just look for entities impacted by this modification. Moreover, they focus on changes on class and method level rather than a block-of-code. They assume a set of predefined system modifications. The problem of capturing and locating specific system entities to change has been addressed by other system reengineering efforts: concept location [4], design patterns [5], aspects [6], and source code evolution [7]. However, these efforts are limited in their expressiveness, formality, and identification precision, as most of them targeted to help program understanding rather than supporting actual program modification. Propagating change modifications on the target system has often been realized using AOP techniques to support software maintenance [8], re-engineering [6], and refactoring [9]. We introduce the concept of re-engineering aspects, or “re-aspects” for short, as a novel, integrated and systematic solution to the system re-engineering and maintenance problem. A re-aspect captures signature of system entities that need to be modified to effect a given change request, actions required (add new code, remove existing code, modify code, or replace code), and code to apply on the located matching entities. Then we automatically analyse the system source code, locate entities that match the specified signatures, conduct detailed impact analysis to identify the impacted entities, and propagate the change required on source code. Re-aspect signature may be class, method, or even code blocks. We introduce two novel signature specification designators to capture formal and flexible semantic and syntactic signatures.

2. RE-ASPECTS
Once a new change request (CR) is received, a change request management process is initiated. This process is conducted to figure out, and document, the expected impacts on system entities. First, it starts with an impact analysis task. The output of it is maintained in a “change set”. Each item in the change set may be removed, modified, replaced, or new code injected. Next, a deeper analysis is then conducted to identify the “impact set” which includes items that will be impacted by changes in the “change set”. Finally, system developers propagate the specified system modifications on the system entities. This might require modifying code developed in different programming languages and different syntactical format (variable name, conditions’ order or format).

2.1 Re-aspects Motivation
Figure 1 shows different code snippets, from an internally developed ERP application called Galactic, vulnerable to different security issues including: (A) built-in security functions that need to be disabled; (B) code vulnerable to authentication bypass attack; and (C) code vulnerable to improper authorization attack. In this figure we also show some possible mitigations (system modifications) to address these vulnerabilities (the code with grey shading) i.e. the required re-engineering of the original code to address the identified security vulnerabilities found.
2.2 Re-aspect Syntax

A re-aspect specifies a single system modification to be applied on the target code base. A re-aspect has a signature, an advice and an action. A re-aspect signature defines footprint of a target system entity that should be deleted/modified/replaced or into which new code is inserted – this may be a line of code, a method, or a class. A re-aspect instance is a matched system entity that matches a given re-aspect signature. Each re-aspect instance maintains their specific context information. A re-aspect action specifies what to do on the re-aspect’s instances. An action may be applied on re-aspect level (i.e. on all re-aspect instances) or on specific instances. A re-aspect impact set represents system entities that will be impacted by a given system modification.

Based on the re-aspect action type, we have four possible re-engineering “re-aspects” types: adding re-aspect: this equates to a conventional AOP code injection aspect. Code to be injected is specified in a separate advice that is weaved with the target system at a given re-aspect instance. It can add any static structure (new method, field, and lines-of-code) to system entities. An anti-aspect has only signature and no advices. The identified code blocks - re-aspect instances - are removed from the target system. A replacing re-aspect is a combination of deletion and adding-aspect. It includes signature of code to be deleted and an advice to be injected. Finally, a modifying re-aspect is the most complicated re-aspect. It makes use of the identified re-aspect instance code to allow the aspect developer to specify selective deletion, reordering, or addition of new nodes into the identified code instance. For example, the problem in Fig. 1-B (Authentication bypass) could be mitigated using a modifying re-aspect advice, as shown in Figure 3. It receives a re-aspect instance (an AST node) as input parameter. At weaving time, we call the modifying aspect script on each identified instance. The returned, modified AST is used to replace the original sub-tree.

2.3 Re-aspect Signature Designators

Supporting system reengineering requires a powerful signature specification approach. Our re-aspect concept is supported with a hybrid approach that delivers flexible syntactical code signature as well as OCL-semantic signature specification designators.

A. Code Snippet Signature Designator: using this designator, developers can specify a flexible code snippet as the aspect signature. Figure 4 shows the template of syntactical code snippet as a signature. Developers use this template to write code parts they are interested in. The flexibility comes when specifying signatures to be matched with code blocks inside methods’ body. A developer can specify the code block they are interested to locate. If the developer does not know the details of the code block, they can use the dummy keyword. This indicates that all statements in the method body will not be considered until a match between the target method statements and the next statement in the given signature is found in the method body.

The syntactical code snippet approach is similar to regular expressions in their expressiveness. Our code snippets have an edge in their matching approach. Regular expressions depend on lexical pattern matching that suffers from lexical problems such as new lines, tabs, brackets, etc. As our code snippet matching is done on Abstract Syntax Trees this avoids such lexical problems and can even match code snippets from different programming languages.

B. Semantic OCL-based Signature Designator: to support more formal semantic re-aspect signatures we use the Object Constraint Language (OCL) as a signature definition language. This is more formal, familiar, and extensible. To enrich OCL with object-oriented programs semantics, we have developed a system-description class diagram, shown in Figure 5. This shows every entity existing in any given object oriented system including component, class, instance, method, inputs, sources, if statements, loops, etc. Moreover, it helps in validating OCL constraints and can be easily extended to capture more abstract system entities and relations such as security APIs, system models (feature, architecture, deployment, design, and testing…). Figure 6 shows examples of OCL-based re-aspect’s signature: (A) get all public methods whose classes implement a specific system feature; (B) get all methods that call a security function.
2.4 Locating Re-aspect Instances

Given a re-aspect signature, to locate the possible re-aspect instances in a target application code base, we first parse the input code and build an abstract syntax tree (AST) representation. This step helps avoiding spacing, comments, brackets and parentheses ambiguities. Moreover, it helps avoid syntax details relevant to different programming languages. Given the source code AST and re-aspects’ signatures, the re-aspect locator traverses the AST looking for matches using one of two matching algorithms. The selection of matching algorithm depends on the given re-aspect signature type. If code snippet then algorithm 1, else algorithm 2.

Algorithm 1, Figure 7: the aspect locator traverses the input source code AST and the given re-aspect code snippet AST looking for matches. The matching takes into consideration the node hierarchy in both the signature and the system code. It treats the dummy constructs as “do not care” nodes in the AST.

Algorithm 2, Figure 8: is based on compiling and validating the given OCL signature using an OCL parser against the system meta-model from Figure 5. Then we generate a visitor class from the given re-aspect OCL signature. The visitor class implements handler methods for every node type specified in the OCL signature. If a visited node has a handler, this handler is called – e.g. a visitor for example (B) Figure 5 will have handlers for method definition and invocation expression nodes. In the invocation expression the visitor will have a condition to check if the invoked method is marked as a security function, then adds this method to the returned list of methods.

2.5 Change Impact Analysis

In AOP the code to be injected is encapsulated in an advice separate from the target cut-point itself. Thus no impact analysis is required. However, with reengineering aspects we have more complicated scenarios where we cut different code parts that have similar signature but different structure and format, are from different places (may have different impact sets), and may be added or modified code. Thus any given system modification requires a detailed impact analysis to identify other system entities that should be updated as a part of given modification.
For each re-aspect instance, we compute a change impact set based on the re-aspect instance type (class, method, property, field, line-of-code). A given system modification will have either local impact or global impact based on re-aspect instance, as follows:

**Lines-of-code:** Has a local impact – i.e. no other system entities will be impacted, thus the change impact set is empty.

**Method:** Has a global impact. To compute the impact set, we locate methods and properties that contain call statement to the modified method. Figure 9-A shows sample OCL expression to locate methods that contain invocation to the modified method.

**Class:** Has a global impact. The change impact set contains all methods that have identifiers of this class (Figure 9-B); properties of this type or have identifier of this type; fields of this class type; and classes that have this class as base class.

**Property:** Has a global impact. To compute the change impact set, we locate all methods that have this property in any expression statement – e.g. assignment, call, if condition, loop statements.

The change impact sets’ entities are located using pre-specified OCL expressions (Figure 9), configured according to re-aspect instance type and name. This avoids building Dependency Graphs (usually adopted by existing approaches and time consuming).

![Figure 10. Control and Data flow analysis, local impact analysis](image)

### 2.6 Change Propagation

The propagation of a given system modification (re-aspect) is straightforward. It depends on the re-aspect type (action) either to insert, delete, replace, or modify code of the located re-aspect instance. All re-aspect instances are updated in the code AST and then code is regenerated with the final updates. For entities in the change impact set of each re-aspect instance (identified in the previous step), we apply the impact re-aspect included in the re-aspect definition, as shown in Figure 2.

Confirming that changes caused by a re-aspect didn’t cause any other problem is an extremely hard problem that requires a deep understanding of the logic behind the code block. Here we focus on confirming that the added, removed, replaced, or modified code does not break the data flow or the control flow of the method, as shown in Figure 10. Control flow analysis (CFA) is used to confirm that the modification does not lead to unreachable code (case 4). Data flow analysis (DFA) confirms that the required data for the modified block are available from previous blocks and that next blocks still have required data items (cases 1, 2, 3).

### 3. EVALUATION

We evaluated the capabilities of re-aspects in locating and propagating a variety of system modifications. Table 1 summarizes the results of using re-aspects to locate matches of a given re-aspects’ signatures using our benchmark applications with a set of three system modifications (from Figure 1) and propagating given changes on the identified matches. We use **precision and recall** metrics to assess our approach effectiveness. From our experiments, the precision of the code-snippet approach is (90%), while its recall is (70%). The precision of the OCL-based approach is (93%) while the recall rate is (87%). The precision of the change propagation module is 88%.

### 4. SUMMARY

We described a novel solution - the "re-aspect" - to the system maintenance problem. A re-aspect captures details of system modifications including signatures of entities that need to be modified; actions to apply on located matches possibly take away (de-weaved), replace, modify or new code inserted; and code to update these entities. A key strength of our re-aspects comes from the signature specification designators. Re-aspect supports two signature specification approaches: code snippet templates, and OCL-based signatures. Re-aspects ease and automate the reengineering process starting with locating system entities to be modified, change impact analysis, and finally propagating updates on located entities. We have validated our approach effectiveness in locating entities to be modified and propagating changes using a set of open source .NET benchmark applications.

**ACKNOWLEDGEMENTS**

The authors are grateful to Swinburne University of Technology and the FRST SPPI project for support for this research.

**REFERENCES**


### Table 1. Results of validating re-aspect to locate and propagate given signatures

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>KLOC</th>
<th>Files</th>
<th>Classes</th>
<th>Authn.</th>
<th>Bypass</th>
<th>Improper Authn.</th>
<th>Sec. Disabling</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C O A M</td>
<td>C O A M</td>
<td>C O A M</td>
<td></td>
</tr>
<tr>
<td>Galactic</td>
<td>16.2</td>
<td>99</td>
<td>101</td>
<td>3 3 3 3</td>
<td>4 7 9 7</td>
<td>3 3 3 3</td>
<td></td>
</tr>
<tr>
<td>SplendidCRM</td>
<td>245</td>
<td>816</td>
<td>6177</td>
<td>- 8 8 8</td>
<td>2 3 1 6</td>
<td>11 11 13 13</td>
<td></td>
</tr>
<tr>
<td>KOBOOB</td>
<td>112</td>
<td>1178</td>
<td>7851</td>
<td>- 6 9 13</td>
<td>8 11 11 11</td>
<td>11 11 11 11</td>
<td></td>
</tr>
<tr>
<td>NopCommerce</td>
<td>442</td>
<td>3781</td>
<td>5127</td>
<td>- - - -</td>
<td>0 1 3 2</td>
<td>10 10 10 10</td>
<td></td>
</tr>
<tr>
<td>Bag/Trace</td>
<td>10</td>
<td>19</td>
<td>298</td>
<td>- - - -</td>
<td>0 1 2 2</td>
<td>7 7 7 7</td>
<td></td>
</tr>
</tbody>
</table>

C: using code snippet, O: using OCL, A: actual instances, M: Successfully Modified