Inference-based Detection of Architectural Violations in MVC2

Shinpei Hayashi, Fumiki Minami and Motoshi Saeki
Department of Computer Science, Tokyo Institute of Technology, Ookayama 2–12–1, Meguro-ku, 152–8552, Tokyo, Japan

Keywords: Architecture Pattern, Code Smell, Program Dependence Graph.

Abstract: Utilizing software architecture patterns is important for reducing maintenance costs. However, maintaining code according to the constraints defined by the architecture patterns is time-consuming work. As described herein, we propose a technique to detect code fragments that are incompliant to the architecture as fine-grained architectural violations. For this technique, the dependence graph among code fragments extracted from the source code and the inference rules according to the architecture are the inputs. A set of candidate components to which a code fragment can be affiliated is attached to each node of the graph and is updated step-by-step. The inference rules express the components' responsibilities and dependency constraints. They remove candidate components of each node that do not satisfy the constraints from the current estimated state of the surrounding code fragment. If the current result does not include the current component, then it is detected as a violation. By defining inference rules for MVC2 architecture and applying the technique to web applications using Play Framework, we obtained accurate detection results.

1 INTRODUCTION

Software architecture patterns (hereinafter, *architecture patterns*), which define the underlying structure of software systems, have been proposed (Buschmann et al., 1996). An architecture pattern comprises multiple components. By defining the responsibilities and dependency constraints for each component, an architecture pattern gives non-functional quality characteristics such as modifiability and reusability to the architecture followed by the pattern. Adoption of an architecture according to an architecture pattern is important for reducing maintenance costs.

To realize characteristics obtained using an architecture pattern, it is necessary for developers to write code for each component according to their responsibilities and dependency constraints. Their violations might degrade the characteristics and benefits of the architecture pattern. However, in practice, the code often violates the component responsibilities and dependency constraints of components. Actually, writing and maintaining code in a proper manner is often burdensome for developers. In particular, developers tend to write inappropriate code because they assign priority to a release as early as possible because of deadline restrictions. In such a case, to gain benefit from the following architecture pattern at the maintenance stage, refactoring (Fowler, 1999) is necessary to mitigate these smelly codes.

Architecture-adapted refactoring techniques based on dependency constraints between components have been proposed already (Hickey and Ó Cinnéide, 2015). However, refactoring techniques based only on dependency constraints might engender another smelly code related to their responsibilities. Refactoring according to architecture patterns demands consideration of both the responsibilities and dependency constraints of components.

In this paper, we aim to support refactoring activities for architecture adaptation with consideration of both the responsibilities and dependency constraints of components in a fine-grained manner. We propose a technique to detect code fragments incompliant to the architecture as fine-grained architecture smells. In the technique, the dependence graph among code fragments extracted from source code and the inference rules according to the architecture are the inputs. The candidates of components to which a code fragment can be affiliated are attached to each node of the graph and are updated step-by-step. The inference rules express the components' responsibilities and dependency constraints, and the rules remove candidates of each node that do not satisfy the constraints using the current estimated state of the surrounding code fragment. If the current result does not include the current component, then it is detected as a violation. In this paper, Model-View-Controller (MVC) Architecture for Web Application (MVC2) (Turner

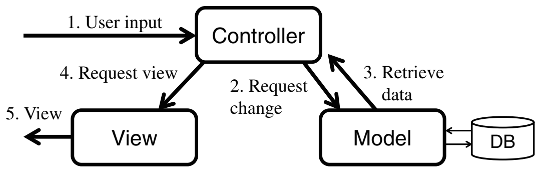


Figure 1: MVC2 architecture.

and Bedell, 2002) and the Play Framework[1] v1 are used as the architecture pattern and its implementation framework, respectively. By defining rules for the MVC2 and applying the technique to web applications using the Play Framework, we obtained accurate detection results.

The remainder of this paper is organized as described below. Section 2 explains architecture patterns and their problems in refactoring using an example. Section 3 discusses related work. Sections 4 and 5 present our proposed technique and its implementation. Section 6 evaluates the technique. Lastly, Section 7 concludes this paper.

2 BACKGROUND

2.1 Architecture Patterns

Architecture patterns (Buschmann et al., 1996) define the underlying structures (*architecture*) of software systems. An architecture consists of encapsulated functional units called *components*. The related architecture pattern guarantees various non-functional characteristics by assigning responsibilities and dependency constraints to the respective components.

MVC2 (Turner and Bedell, 2002) is an architecture pattern, which is an extended version of the MVC pattern suitable for web applications.

The structure of MVC2 is portrayed in Figure 1. In MVC2, *Model* is responsible for domain logics including database operations, *View* is for presentation logics, and *Controller* is for these controls according to the user inputs. In the processing flow in MVC2 is the following. First, the *Controller* component receives user input. *Controller* requests data changes to the *Model* component if necessary, and *Model* updates the data in the database. Then, *Controller* requests the data required for display from *Model*, and *Model* retrieves the corresponding data from the database and passes them to *Controller*. Finally, *Controller* passes the data to the *View* component and requests that they be displayed.

Dividing the respective responsibilities makes it easy to replace each component. For example, it is possible to change the appearance of an application by replacing *View* and to execute automated tests by replacing *Controller*. Another benefit is that developers can specifically examine particular concerns.

MVC2 also defines the relation of accessibility between components. For example, *Model* cannot refer to *View*. This restriction can be regarded as a dependency constraint on components, which provides benefits in maintaining the application. For example, preparing *Model* not depending on *View* presents the advantage that changes of the application appearance do not propagate to the domain logics.

2.2 Problems in Adapting Architecture Patterns

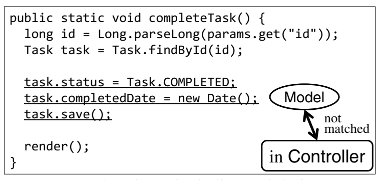
An architecture pattern guarantees various non-functional quality characteristics to the following architecture. This is useful for reducing maintenance costs. However, these characteristics are guaranteed only when the code of each component correctly follows the responsibilities and dependency constraints of the component; they are lost when the code violates the structure defined by the pattern. We categorized the violations in an architecture pattern into the following two types.

- *Violation of the responsibilities of components*. It occurs when the role of a code fragment in a component embodies the responsibilities of another component, e.g., a presentation logic, which should be described in *View*, is written in *Model* or *Controller*.
- *Violation of the dependency constraints of components*. It occurs if an unauthorized dependency exists between code fragments, e.g., a code fragment in *Model* refers to a field or method in *View* or *Controller*.

Violations of these two types stand on different viewpoints. Therefore, a refactoring to resolve violations of one type might cause those of the other type. A mechanism is required for finding refactoring opportunities that correctly resolve violations of both types.

An example of violations in MVC2 is presented in Figure 2(a). This example includes an action method described as *Controller* for completing a task in a task management application developed using Play framework. After specifying a task object based on the input from the user, this method updates the completion status of the task, updates the completion date, and saves the task among its code fragments, shown as underlined. These update and save are inseparable series of a procedure related to a database update,

[1]https://www.playframework.com/

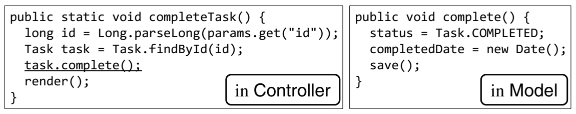


```
public static void completeTask() {
  long id = Long.parseLong(params.get("id"));
  Task task = Task.findById(id);

  task.status = Task.COMPLETED;
  task.completedDate = new Date();
  task.save();

  render();
}
```

(a) Code snippet including a violation.



```
public static void completeTask() {
  long id = Long.parseLong(params.get("id"));
  Task task = Task.findById(id);
  task.complete();
  render();
}
```

```
public void complete() {
  status = Task.COMPLETED;
  completedDate = new Date();
  save();
}
```

(b) Refactored version.

Figure 2: Example of violation in architectural constraint.

and they can be regarded as domain logics for which *Model* should own their responsibility. However, because this method belongs to *Controller*, a gap separates the role and responsibility of the method, representing a violation of the responsibility.

This violation can be mitigated by refactoring to the structure shown in Figure 2(b). The corresponding code fragments are extracted as an individual method *complete* by Extract Method. It is moved to *Model* by Move Method so that it no longer has a mismatch in its responsibility. To realize such refactoring, it is necessary to identify code fragments violating the architectural constraints.

3 RELATED WORK

Budi et al. proposed a violation detection technique for multilayer architecture using machine learning and the accessibility relation between layers (Budi et al., 2011). In this technique, classes are classified into layers using machine learning from the basic information of classes. Violations are detected by comparing the accessibility relation of classes and those between layers. In addition, Hickey and Ó Cinnéide proposed a search-based refactoring technique of multilayer architecture (Hickey and Ó Cinnéide, 2015). This technique uses metrics measuring access violations between layers as an evaluation function and refactorings as transitions in the search space, and finds appropriate states. Although these techniques use dependency constraints, they depend on the original code and training data related to responsibilities. Such techniques differ from ours in that they do not directly address the architectural responsibilities.

ArchFix (Terra et al., 2015) detects architectural violations and recommend refactoring operations to repair the detected violations. Macia et al. proposed a technique to detect code anomalies using architecturral concern-based metrics (Macia et al., 2012; Macia et al., 2013). Although these techniques utilize dependencies to detect violations or anomalies, they do not utilize statement-level dependencies to infer the roles of code fragments and detect violations on them. Such an approach is effective when refactoring controller methods including statements of different roles mixedly.

A sequence of refactoring operations is needed after detecting a violation. Tsantalis and Chatzigeorgiou proposed a technique to improve maintainability by Move Method refactoring and implemented it as JDeodorant (Tsantalis and Chatzigeorgiou, 2009). JDeodorant confirms the improvement of maintainability by Move Method by measuring coupling and cohesion metrics. In addition, Sales et al. demonstrated the possibility of automated refactoring with Move Method with higher accuracy using the similarity of dependency sets (Sales et al., 2013). Trifu and Reupke discussed relationship between a design flaw and the number of directly observable indicators (Trifu and Reupke, 2007). They defined specifications of design flaws including context and indicators, and a diagnosis strategy using indicators and correction strategies written in a natural language. They also presented a tool to identify design flaws. Their indicators for design flaw identification are defined as a combination of design metrics and structural information. ClassCompass (Coelho and Murphy, 2007) is an automated software design critique system, and it has a feature to suggest design correction based on rules written in a natural language. These techniques differ from the proposed technique in that they do not consider architectural constraints.

4 PROPOSED TECHNIQUE

4.1 Overview

For taking both responsibilities and dependency constraints of components into account in detecting architectural violations, the proposed technique uses a role inference. In this paper, a (possible) *role* of a code fragment is a set of components to which the code fragment can belong. The role inference infers the components to which each code fragment can belong using *inference rules* based on the responsibilities and dependency constraints of components.

An overview of the proposed technique is presented in Figure 3. Its inputs are the source code, domain knowledge for initializing the role of code fragments, and an inference rule database. The technique first analyzes the given source code and builds a program dependence graph by extracting code fragments

Table 1: Dependencies among code fragments.

Relations	Depender	Dependee	Definition
Def-Use	Statement	Statement	Defining and referring a variable
Access	Statement	Field	Reading and writing a field
Invocation	Statement	Method	Invoking a method
Inclusion	Statement	Method	Inclusion of a code fragment

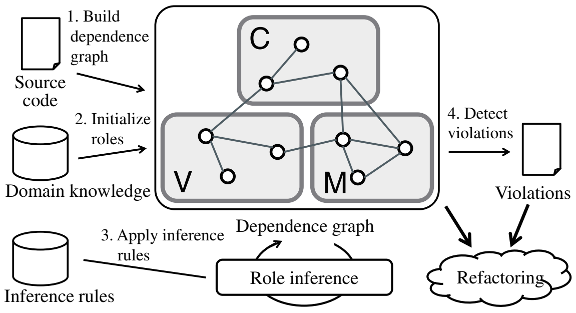


Figure 3: Overview of the proposed technique.

and relations among them (Step 1, Section 4.2). Next, it initializes the role of each code fragment based on the domain knowledge (Step 2, Section 4.3). Then, the role inference is performed to narrow down the possibility of components using inference rules (Step 3, Section 4.4). This process identifies code fragments that can belong to certain components. After the role of each code fragment is determined, violations are detected by comparing the inferred role with the current belonging component (Step 4, Section 4.5). For a code fragment, a violation is detected if the current component of the fragment is not included in the role of the fragment. We regard such violations as fine-grained architectural smells. Candidate refactoring operations to solve these smells are Extract Method, Move Method (Fowler, 1999), etc. Currently, the proposed technique does not include the derivation of refactoring operations.

The proposed technique requires the preparation of domain knowledge and inference rules. We can prepare them in advance if we use a framework. Once experts of a specific framework build domain knowledge and inference rules, non-expert framework users can reuse them.

4.2 Building Dependence Graph

In the technique, a dependence graph is built by extracting code fragments from the source code and their mutual relations. We use sentence-level code fragments, which are appropriate for extracting relations. Also, for taking the method invocation into consideration, method invocations in sentences are handled as individual fragments. In addition, fields and methods are acquired as nodes.

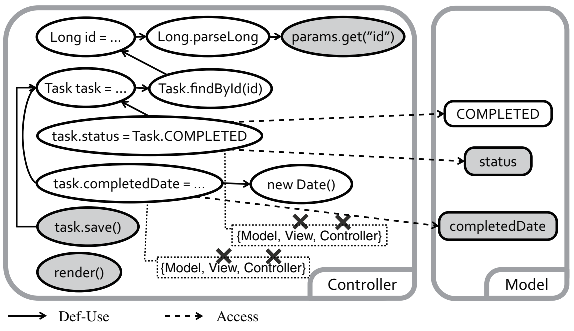


Figure 4: Dependence graph and role inference in the example in Figure 2(a).

Table 1 shows the relations to be extracted. When referring to the variable *v* defined in a certain code fragment in another code fragment, a *Def-Use* dependence on *v* is assumed between the two fragments. When reading or writing a certain field in a certain code fragment, an *Access* dependency from the code fragment to the field is assumed. When invoking a certain method in a certain code fragment, an *Invocation* relation from the fragment to the method is assumed. An *Inclusion* dependency is defined between a code fragment and a method when the fragment is included in the method.

Figure 4 portrays a dependence graph built from the code shown in Figure 2(a). We can find that code fragments corresponding to each sentence or method invocation are extracted as nodes. In addition, dependencies between nodes are defined; for example, for the node "*Task task* = ⋯" which defines the variable *task*, several nodes including "*task.status* = *Task.COMPLETED*" and "*task.completeDate* = ⋯" are defined as using the variable (Def-Use).

4.3 Initializing Roles

Each node in the dependence graph has its particular role. In the example of Figure 4, *Model*, *View*, and *Controller* are the target components. Fundamentally, we assign all possibilities to each node, i.e., the role of all nodes is initialized as a set of all components. However, we narrow the role of some nodes based on the domain knowledge.

In the example presented in Figure 4, all the possibilities $\{Model, View, Controller\}$ are allotted to the white nodes. In contrast, the roles of gray nodes are

Table 2: Modifiability relation in the Modification rule.

From\To	*Model*	*View*	*Controller*
Model	√		
View		√	
Controller	√		√

specified uniquely by the domain knowledge. For example, in Play Framework v1, invocations of the method *render()* are well-known to be located in controllers. Therefore, a role of $\{Controller\}$ is allotted to the associated node. In addition, because the fields *status* and *completedDate* in the classes in *Model* behave as models, their roles are initialized as $\{Model\}$. In this way, most of the input domain knowledge functions as a dictionary of method names and their corresponding components.

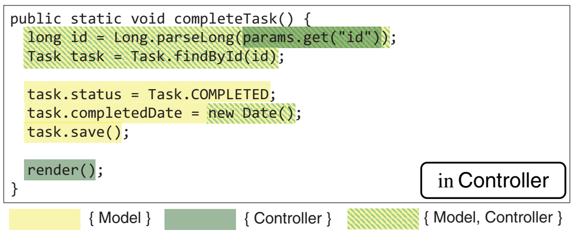
4.4 Applying Inference Rules

In the role inference, candidate components in a role are narrowed down gradually. To update the role of each node, we use the inference rules representing the responsibilities or dependency constraints of components. An inference rule removes inappropriate candidate components from a role by examination of the dependencies among nodes and the roles of their neighboring nodes in the graph. The update of a role might influence another; inference rules are repeatedly applied until no rule produces a change.

An example of role inference by Modification rule in MVC2 is the following. The Modification rule is based on the dependency constraint of *modifiability* in MVC2 components. In MVC2, the state of *Model* can only be modified by *Model* and/or *Controller*: not by *View*.

This constraint can be represented as a binary relation of components shown in Table 2. Each row and column of the table respectively represent the source and target components of modification to which the focused code fragments belong. The symbol $\surd$ in the table denotes the possibility of modifications. We exclude as inappropriate those possibilities of candidate components in a role which do not satisfy this constraint (cells without $\surd$).

Review of Figure 4 shows how Modification rule is applied. The role of the field "*status*" is $\{Model\}$ based on the domain knowledge. In addition, the dependencies express that the statement node "*task.status = Task.COMPLETED*" accesses the field "*status*". Because this statement node modifies the field node determined as *Model*, it is apparent that the statement node should not be *View* based on Table 2. Therefore, the possibility of *View* is excluded from the role of the statement node, and the possibility of



```
public static void completeTask() {
  long id = Long.parseLong(params.get("id"));
  Task task = Task.findById(id);

  task.status = Task.COMPLETED;
  task.completedDate = new Date();
  task.save();

  render();
}
```

Figure 5: Result of role inference.

$\{Model, Controller\}$ remains. In the same way, *View* is excluded from the role of node "*task.completeDate* $= \cdots$". In addition to these, the possibility of *Controller* was also excluded by another rule named Anemic Domain Model, and the possibility of the two nodes is eventually limited to *Model*. Figure 5 shows the final result of the role inference of the example shown in Figure 4.

4.5 Detecting Violations

The inferred roles of code fragments are compared with their currently belonging component to detect violations. If the currently belonging component is not included in the inferred role, then it is detected as an architectural violation that the code fragment is incompliant to the responsibilities, dependency constraint, or both.

In the example shown in Figure 4, the possibility of *View* and *Controller* is excluded from the role of nodes "*task.status = Task.COMPLETED*" and "*task.completeDate* $= \cdots$" by the role inference, and their role is finally specified as *Model*. However, these nodes currently belong to *Controller*. Because the belonging component is not included in the inferred role, it is detected as a violation. In this case, applying Move refactoring to move the corresponding code fragment to *Model*, which is the remaining candidate component in the role, is considered.

5 IMPLEMENTING OUR TECHNIQUE

5.1 Inference Rules

We have defined inference rules for MVC2 with taking both the responsibilities and dependency constraints of components into consideration. The rules on dependency constraints were Def-Use expressing constraints on data dependence, Visibility expressing constraints on interface visibility, and Modification expressing constraints on availability of data changes. They were easily derived from the definition of MVC2, and we believe that rules based on similar constraints in some architectural patterns can be defined in the same way. For example, the Layers architecture (Buschmann et al., 1996) restricts accesses of non-adjacent layers, which can be defined as dependency constraints.

Meanwhile, the definition of responsibilities in MVC2 was ambiguous, and it was difficult to compose rules on them based only on the definition. Therefore, rules on responsibilities were extracted based on the violation patterns observed in actual projects using MVC2. One of the authors manually analyzed the source code of 11 web applications developed by students majoring in computer science and identified the code fragments incompliant to the responsibilities of the components. Then, two patterns could be observed.

The first pattern is generating strings only for display in *Controller*. This expresses a situation that statements generating strings to be displayed only, which should be regarded as presentation logic, are in the *Controller* component instead of the *View* component. It violates the architecture of MVC2. In order to be compliant to MVC2, only the source data of such strings should be passed from the *Controller* component to the *View* component, and the generation of strings to be displayed should be performed in the *View* component.

The second pattern is defining domain logics in *Controller*. This is related to Anemic Domain Model anti-pattern (Fowler, 2003), which is a case in which domain logics are in *Controller* instead of *Model*. In order to be compliant to MVC2, it is necessary to describe the domain logic in the *Model* component as much as possible.

Including these two related to the responsibilities of components, we defined totally five inference rules for detecting architectural violations in MVC2 as follows.

- Def-Use uses the data dependency among components, e.g., variables defined in *View* cannot be referred by *Model* or *Controller*.
- Visibility excludes the candidate components in roles using the relation among components and the access and invocation dependencies.
- Modification excludes the candidate components in roles using the relation among components and the access, invocation, and inclusion dependencies.
- Visual String looks for code fragments in *Controller* that generate strings to be displayed only in *View*. This rule obtains the interface of *View* from the domain knowledge and finds code fragments generating strings to be passed to *View* but not to *Model* by tracing *Def-Use* dependency, and not to be affected by control flow. More specifically, it checks paths on the data and control flows. If a path from a string expression s generated in *Controller* to a node in *View* is found, but not path from s to nodes in *Model* or any branch conditions, the role of s is assigned as $\{View\}$.
- Anemic Domain Model explores a set of code fragments in *Controller* for which dominant dependees act as *Model* or not and determines the role of the code fragments also as *Model*. More specifically, it counts the dependees having the role of *Model* and those of *Controller* for each node in *Controller*. If the number of dependees in *Model* is greater than those in *Controller*, the role of the node is assigned as $\{Model\}$.

5.2 Implementation

We have implemented the proposed technique as an Eclipse plug-in as well as the five inference rules. Domain knowledge of web applications using Play Framework has been predefined, which is used for initializing and inferring the role of each code fragment and for the Visual String rule.

For building dependence graphs, we used jxplatform[2]. The jxplatform is a static analysis tool for Java to build a Java model consisting of system dependence, program dependence, control flow, and call graphs.

6 EVALUATION

6.1 Study Design

We evaluated our technique by application of the implemented detector to multiple projects. In the evaluation, we focused on two criteria: *accuracy* of the detection results and *validity* of the inference rules. The first criterion relates to the possibility of detecting code fragments that violate the responsibility or dependency constraint of components expressed by each inference rule. The second criterion is used to ascertain whether each inference rule is actually meaningful, or not. We confirmed that each inference rule actually excludes candidates from roles and that such removal of candidates actually affects the detection of violations.

We applied our technique to 37 web application projects developed by students majoring in computer

[2]https://github.com/katsuhisamaruyama/jxplatform

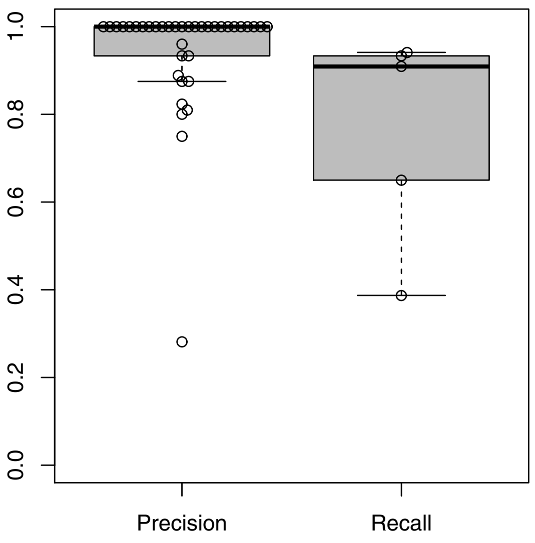


Figure 6: Distribution of precision and recall values.

science.[3] Each project uses Play Framework v1 and follows MVC2 architecture. Regarding the accuracy, we used two metrics: precision and recall. In measuring the precision, one author confirmed the detected violations and judged their correctness for all 37 projects. In measuring the recall, because it was difficult to prepare the correct detection results of all projects, one of the authors manually identified all violations for five randomly sampled projects. The recall values were measured for these five projects. Regarding the validity, we counted applications for each inference rule. We also counted only those which contributed to the detection of violations.

6.2 Results

Figure 6 shows the distribution of the precision and recall values together with a bee swarm plot including all the data points. Many projects showed high precision and recall values; their averages are respectively 0.94 and 0.76. Since we have obtained both high precision and recall rates in many projects, we can conclude that the proposed technique could detect violations from the subject projects. However, some projects showed very low values. We investigated that phenomenon and reached the conclusion that many false positives were produced by Anemic Domain Model rule. As a result of faulty decisions that the role of a code fragment is decided as *Model*, the technique produced subsequent wrong decisions that the peripheral code fragments similarly decided as *Model* falsely, which increased the incidence of wrong results. In addition, when examining the projects with a low recall value, results showed that some violations based on Anemic Domain Model were not detected. This result derived from the failure of the propagation in role inference attributable to the long distance between the violated code fragments and their related fragments.

[3]The projects used by the analysis in Section 5.1 were excluded.

Table 3: Numbers of applications of inference rules.

Rule	# apps.	# effective apps.
Def-Use	3578	315
Visibility	781	583
Modification	925	478
Visual String	48	48
Anemic Domain Model	856	620

Table 3 shows how many inference rules were applied. The columns indicate the name of the inference rules, the number of applications that succeeded to exclude at least one candidate of a role, and the number of *effective* applications, which succeeded to exclude at least one candidate in a role and which affected the detection of at least one violation. The effective applications were numerous for all the inference rules. This result indicates that all the inference rules influenced the detection of violations and were effective.

6.3 Threats to Validity

Internal Validity. Although the precision and recall values were measured based on the number of violated code fragments, the cause of multiple violations might be the same. However, it is difficult to define clear criteria to identify the causes. Also, the oracle preparation was done by one of the authors, which might be biased. Similarly, there might have been ambiguity in deciding the roles of the oracle in relation to the Anemic Domain Model. Preparation of more reliable benchmarks is necessary. Additionally, since two inference rules were derived using projects by students, which are the same sort of the ones used in the evaluation, it might introduced an overfitting to detect violations in the same sort of projects.

External Validity. All projects in our evaluation were developed by students, which might result in different results when we apply our technique to business applications in general developed by practitioners. Additionally, whether our technique works for other frameworks, or not, has not been investigated.

7 CONCLUSION

To perform refactoring to adapt a program for an architecture pattern, a technique considering both the responsibilities and dependency constraints of components is required in a fine-grained way. As described herein, we proposed a technique to detect violations by introducing role inference rules to estimate the components to which each code fragment can belong. In role inference, both the responsibilities and dependency constraints of components are expressed as inference rules. Using MVC2 and Play Framework as the target architecture and framework, we have implemented an automated violation detector and evaluated its usefulness by its application to multiple projects.

An important future task is to establish a refactoring technique of the detected violations. In this paper, we did detect the code fragments including violations, but we did not address to which technique the detected code fragments should be moved. It is preferable to move a code fragment containing violations along with its surrounding related code fragments. It is important to find the code fragments to be moved at the same time, which can be specified using the result of our role inference. In addition, it is also necessary to find other refactoring techniques to fix violations that are unable to be fixed by Move refactoring.

It is also an important task to apply our approach to other architectural patterns. We believe that the inference rules on dependency constraints are considered to be applicable to other architecture patterns such as Layers, but rules on responsibilities can vary greatly depending on the architecture themselves. It is important to confirm whether inference rules can express responsibilities of components for various architecture patterns.

ACKNOWLEDGEMENTS

This work was partly supported by JSPS Grants-in-Aid for Scientific Research Numbers JP15K15970, JP15H02683, and JP15H02685.

REFERENCES

Budi, A., Lucia, Lo, D., Jiang, L., and Wang, S. (2011). Automated detection of likely design flaws in layered architectures. In *Proc. 23rd International Conference on Software Engineering and Knowledge Engineering*, pages 613–618.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-oriented Software Architecture*. John Wiley & Sons, Inc.

Coelho, W. and Murphy, G. (2007). ClassCompass: A software design mentoring system. *Educational Resources in Computing*, 7:1–18.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Fowler, M. (2003). AnemicDomainModel. http://www.martinfowler.com/bliki/AnemicDomainModel.html.

Hickey, S. and Ó Cinnéide, M. (2015). Search-based refactoring for layered architecture repair: An initial investigation. In *Proc. 1st North American Search Based Software Engineering Symposium*.

Macia, I., Arcoverde, R., Cirilo, E., Garcia, A., and von Staa, A. (2012). Supporting the identification of architecturally-relevant code anomalies. In *Proc. 28th IEEE International Conference on Software Maintenance*, pages 662–665.

Macia, I., Garcia, A., Chavez, C., and von Staa, A. (2013). Enhancing the detection of code anomalies with architecture-sensitive strategies. In *Proc. 17th European Conference on Software Maintenance and Reengineering*, pages 177–186.

Sales, V., Terra, R., Miranda, L. F., and Valente, M. T. (2013). Recommending move method refactorings using dependency sets. In *Proc. 20th Working Conference on Reverse Engineering*, pages 232–241.

Terra, R., Valente, M. T., Czarnecki, K., and Bigonha, R. S. (2015). A recommendation system for repairing violations detected by static architecture conformance checking. *Software: Practice and Experience*, 45(3):315–342.

Trifu, A. and Reupke, U. (2007). Towards automated restructuring of object oriented systems. In *Proc. 12th Working Conference on Reverse Engineering*, pages 39–48.

Tsantalis, N. and Chatzigeorgiou, A. (2009). Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367.

Turner, J. and Bedell, K. (2002). *Struts Kick Start*. Sams.