Poster: Conquering Uncertainty in Java Programming

Takuya Fukamachi*, Naoyasu Ubayashi*, Shintaro Hosoai* and Yasutaka Kamei*
*Kyushu University
Fukuoka, Japan
Email: {fukamachi@posl.ait, ubayashi@ait, hosoai@qito, kamei@ait}.kyushu-u.ac.jp

***Abstract*—Uncertainty in programming is one of the challenging issues to be tackled, because it is error-prone for many programmers to temporally avoid uncertain concerns only using simple language constructs such as comments and conditional statements. This paper proposes *ucJava*, a new Java programming environment for conquering uncertainty. Our environment provides a modular programming style for uncertainty and supports test-driven development taking uncertainty into consideration.**

I. Introduction

Uncertain concerns can appear in all software development phases: uncertain user needs in requirements analysis, vague architectural decisions in design modeling, and alternative algorithms in programming. Recently, uncertainty has attracted a growing interest among researchers [1]. However, uncertainty in programming has not been well explored. Assume the following situations: 1) uncertain whether a portion of a program is really needed or should be replaced by other code in terms of refactoring; 2) uncertain which algorithm should be adopted to realize performance requirements, and 3) uncertain which code is finally used because of changeable stakeholder requirements. We have to temporally comment out the target statements to skip an uncertain concern, insert a superfluous *if* statement to be able to select an alternative uncertain choice, or use conditional compilation with preprocessor directives. After that, we have to test the program. These comments, conditional statements, and preprocessor directives make difficult to understand the program code, because they impede the separation of concerns in terms of modularity. The exploratory modification process may be repeated again and again until all uncertain concerns are fixed. If an uncertain concern cross-cuts over multiple places in a program, the number of comments or conditional statements increases and the version control of the modified code becomes tremendously difficult. Moreover, we may have to return all of the modified portions to the original code or one of the final decided code if the uncertain concern is fixed to be certain. This task is tedious and error-prone. It may become a cause of a meaningless defect. We consider that many programmers have an experience of encountering this kind of problems. Currently, we have neither any method nor tool for conquering uncertainty.

One of the reasons why uncertainty cannot be dealt with in current programming languages is that the state-of-the-art module mechanisms do not regard an uncertain concern as a first-class pluggable software module. If uncertainty can be dealt with modularly, we can add or delete uncertain concerns to/from code whenever these concerns appear or disappear.

This paper proposes *ucJava*, a Java programming environment for conquering uncertainty, that provides a modular programming style for uncertainty and supports test-driven development taking uncertainty into consideration.

II. Overview of ucJava

We extend *Archface* [5], [6], an interface mechanism among models and programs, to support uncertainty. Currently, we are proceeding a research project *"Model-Driven Development Embracing Uncertainty"* under the support of the Grant-in-aid for Scientific Research in Japan [2]. *Archface* plays an important role in this project, because we can explicitly describe uncertainty in requirements, design models, and programs. In this paper, we focus on uncertainty in programming.

A. Uncertainty as Pluggable Interface

Archface, which supports *component-and-connector* architecture, consists of two kinds of interface, *component* and *connector*. The former is the same with ordinary Java interface and the latter defines the message interactions among components. A connector is specified using the notation similar to FSP (Finite State Processes) [4]. An *Archface* definition of the *Observer* pattern is shown in List 1.

```
[List 1] -- Java & FSP-like Syntax
01: interface component cSubject {
02:   public void addObserver(Observer);
03:   public void removeObserver(Observer);
04:   public String getState();
05:   public void setState(String);
06:   public void notify();
07: }
08:
09: interface component cObserver {
10:   public void update();
11: }
12:
13: interface connector cObserverPattern
14:           (cSubject, cObserver){
15:   cSubject = (cSubject.setState->cSubject.notify
16:     ->cObserver.update->cSubject.getState->cSubject);
17:   cObserver = (cObserver.update->cSubject.getState
18:     ->cObserver);
19: }
```

As a representative work, a method for expressing uncertainty using a partial model is proposed in [1]. We apply this idea to *ucJava*. A partial model is a single model containing all possible alternative uncertain designs and is encoded in propositional logic. We can check whether or not a model including uncertainty satisfies some interesting properties.

In *ucJava*, uncertainty is introduced modularly by extending the existing interface as illustrated in Figure 1. The symbols $\{\}$ and $[]$ represent *alternative* and *optional*, respectively.

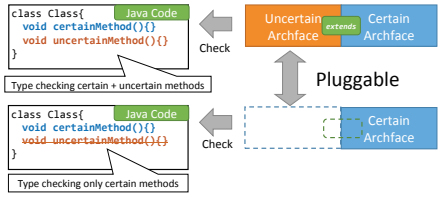

Fig. 1. Pluggable Uncertainty

This notation, which is inspired from software product lines [7], can have an expressive power equal to a partial model. If a programmer doubts whether or not `notify` is really needed for understandability, he or she only has to change *Archface* as shown in List 2. It is also uncertain which of `removeObserver` or `deleteObserver` should be defined; and whether `notify` is called from `setState`.

```
[List 2]
01: interface component uSubject extends cSubject {
02:   public void {removeObserver(), deleteObserver()};
03:  [public void notify();]
04: }
05:
06: interface connector uObserverPattern
07:   extends ObserverPattern( cSubject, cObserver) {
08:   cSubject = (cSubject.setState-> [cSubject.notify]
09:     ->cObserver.update->cSubject.getState->cSubject);
10: }
```

B. Type Checking in ucJava

Uncertainty is a target of compilation in *ucJava* whose type checker verifies not only the conformance of a Java program to its *Archface* but also the consistency among components and connectors. The *ucJava* compiler generates a partial model from *Archface* definitions and verifies whether a Java program is an instance of the partial model. In List 2, type check is passed if a Java program corresponds to either of the following: 1) `notify` is defined and is called in the program; 2) `notify` is not defined; 3) both `deleteObserver` and `removeObserver` are defined and either of them is called in the program; or 4) either `deleteObserver` or `removeObserver` is defined and is called in the program. Otherwise, type checker generates an error. In case of List 2, we can continue the development regardless of whether or not `notify` is defined, because there are no inconsistencies in the *Archface* definition. We can make a program without using comments or conditional statements even if uncertain concerns are contained in the program. We have only to declare an uncertain *Archface* and implement it.

C. Test-Driven Development with Uncertainty

Unfortunately, unit testing tools such as JUnit cannot be applied without making extra test cases taking *optional* and *alternative* into account. We cannot reuse the original test cases and the modification of test cases occurs whenever the specification of uncertainty is changed. This rework is repeated again and again until uncertain concerns disappear.

To deal with this problem, original test cases are automatically modified in *ucJava* using AspectJ to test uncertain methods as illustrated in Figure 2. The call to an optional method can be skipped and the call to a method defined in the original test case can be replaced by an alternative method. In

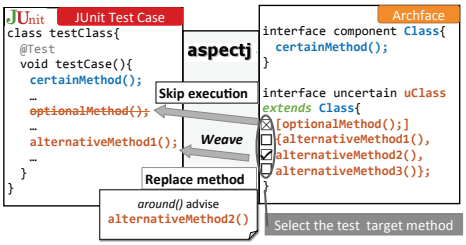

Fig. 2. Automated Test Case Generation and Execution

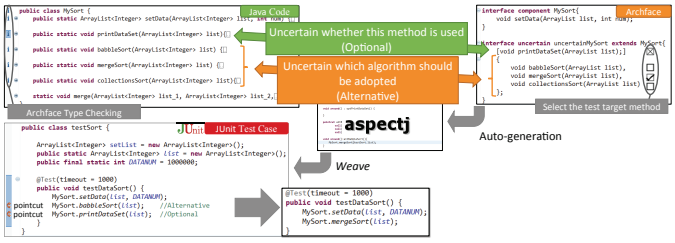

Fig. 3. Programming and Testing in *ucJava*

this case, *around* advice is used. All possible test cases can be covered automatically.

D. Example

Assume that a performance issue *"sort has to be performed within one second"* is tested using a given data set and there are three sort programs: *bubble*, *merge*, *Java Collections.sort*. It is uncertain which algorithm should be selected. Figure 3 illustrates the programming and testing in *ucJava*. In this example, uncertainty is fixed by selecting an algorithm that satisfies the performance constraint.

III. Conclusions and Future Work

The *ucJava* programming environment is the first result of our ongoing project *"Model-Driven Development Embracing Uncertainty"*. As the next step, we plan to pursue the following research items: 1) support of uncertainty in requirements and design modeling; 2) empirical studies using OSS projects; and 3) support of concolic testing [3], a hybrid verification that performs both symbolic execution and concrete testing.

References

[1] Famelis, M., Salay, R., Chechik, M.: Partial Models: Towards Modeling and Reasoning with Uncertainty, In *Proceedings of the 34th International Conference on Software Engineering*, pp.573-583, 2012.

[2] Grants-in-Aid for Scientific Research, http://www.jsps.go.jp/english/e-grants/index.html, 2014.

[3] Koushik, S., Marinov, D., and Agha, G.: CUTE: A Concolic Unit Testing Engine for C, In *Proceedings of the 13th ACM SIGSOFT international Symposium on Foundations of Software Engineering*, pp.263-272, 2005.

[4] Magee, J. and Kramer, J.: Concurrency: State Models & Java Programs second edition, Wiley, 2006.

[5] Ubayashi, N., Nomura, J., and Tamai, T.: Archface: A Contract Place Where Architectural Design and Code Meet Together, In *Proceedings of the 32nd International Conference on Software Engineering*, pp.75-84, 2010.

[6] Ubayashi, N., Ai, D., Li, P., Li, Y., Hosoai, S., and Kamei, Y.: Abstraction-aware Verifying Compiler for Yet Another MDD, In *Proceedings of the 29th International Conference on Automated Software Engineering*, pp.557-562, 2014.

[7] Zhang, H. and Jarzabek, S.: XVCL: A Mechanism for Handling Variants in Software Product Lines, *Sci. Comput. Program.*, vol.53, no.3, pp.381-407, 2004.