A Framework and Tool Supports for Testing Modularity of Software Design

Yuanfang Cai, Sunny Huynh
Department of Computer Science
Drexel University
Philadelphia, PA
yfcai, sh84@cs.drexel.edu

Tao Xie
Department of Computer Science
North Carolina State University
Raleigh, NC
xie@csc.ncsu.edu

ABSTRACT

Modularity is one of the most important properties of a software design, with significant impact on changeability and evolvability. However, a formalized and automated approach is lacking to test and verify software design models against their modularity properties, in particular, their ability to accommodate potential changes. In this paper, we propose a novel framework for testing design modularity. The software artifact under test is a software design. A test input is a potential change to the design. The test output is a modularity vector, which precisely captures quantitative capability extents of the design for accommodating the test input (the potential change). Both the design and the test input are represented as formal computable models to enable automatic testing. The modularity vector integrates the *net option value* analysis with well-known design principles. We have implemented the framework with tool supports and tested aspect-oriented and object-oriented design patterns in terms of their ability to accommodate sequences of possible changes. The results showed that previous informal, implementation-based analysis can be conducted by our framework automatically and quantitatively at the design level. This framework also opens the opportunities of applying testing techniques, such as coverage criteria, on software designs.

Categories and Subject Descriptors: D.2.10 [Software Engineering]: Design

General Terms: Design.

Keywords: Design Testing, Modularity, Changeability

1. INTRODUCTION

Modularity in software designs offers evolvability, which has been long recognized to have enormous technical, organizational, and ultimately economic value. Although various approaches have been developed in previous work to test or verify software design models against specified properties to ensure design correctness, we lack a formal framework to ensure the modularity in designs.

To assess design modularity quantitatively and objectively, we propose a novel framework to test the modularity properties of a software design and develop tool supports to conduct modularity testing. In our framework, the software artifact under test is a software design. A test input is a potential change to the design. The test output is a *modularity vector*, which precisely captures quantitative capability extents of the design for accommodating the test input (the potential change). The modularity vector integrates Baldwin and Clark's *net option value* analysis [1] with well-known design principles, such as maximizing cohesion, minimizing coupling, open to extension, and close to modification [16, 14].

In our framework, both the software under test and the potential changes are formally modeled as *Augmented Constraint Networks* (ACNs) [3, 2]. In an ACN, design dimensions and environmental conditions are uniformly modeled as variables, possible choices as values of variables, and relations among decisions as logical constraints. The ACN modeling formalizes the key notions of Baldwin and Clark's design rule theory [1] and Parnas's information hiding criterion [15, 2]. The ACN modeling makes our framework independent of particular modularity techniques and language paradigms, and enables automatic and quantitative design modularity testing.

Previous research has analyzed software modularity at the level of source code or in qualitative, intuitive, and heuristic ways. For example, Hannemann and Kiczales compared the evolvability and modularity properties of design pattern implementations using the aspect-oriented [10] (AO) versus object-oriented (OO) paradigm. They show the actual code implementing these choices as the evidence of their analysis. However, designers frequently face similar questions before coding. As a feasibility study, we compare the designs of their AO versus OO observer pattern using our framework against sequences of potential changes (as test inputs). The results show that our framework automates Hannemann and Kiczales's evolvability and modularity analysis precisely at the design level, and provides additional insights.

The rest of this paper is organized as follows. Section 2 presents the details of the framework. Section 3 presents our experience of applying this framework to compare design alternatives. Section 4 discusses related work, and Section 5 concludes.

2. FRAMEWORK

Our framework receives two types of inputs: a software design (the software artifact under test) and a potential change (the test input). In this section we introduce the modeling of the two inputs as Augmented Constraint Networks (ACNs) [3, 2], the automatic derivation of modularity properties with and without the change, and the computation of a modularity vector as the test output.

2.1 Modeling of Design and Test Input

An *Augmented Constraint Network* (ACN) was proposed in our previous work [3] as a formal design representation better subject to

ASE'07, November 4–9, 2007, Atlanta, Georgia, USA.

automated analysis of the design evolvability and economic-related properties. The core of an ACN is a finite-domain *constraint network* [13].

A constraint network consists of a set of *design variables* (modeling design dimensions or relevant environmental conditions) and a set of logical constraints (modeling the relations among them). Each *design variable* has a *domain* that comprises a set of *values*, each representing a decision or condition. A design decision or environmental condition is represented by a binding of a *value* from a *domain* to a variable. We use *orig* (short for original) to generally represent a currently selected design decision in a given dimension, and use $other$ as a value to represent unelaborated possibilities.

We model the design under test as an ACN. For example, The following lines are the variables used to model a widely used Figure Editor (FE) system using an observer pattern [8, 6]: the notification policy, update policy, mapping data structure, how the colors of the subjects should be observed, and the five involved classes.

```
1:  spec_notify_policy:{push,pull};
2:  spec_update_policy:{simple,complex};
3:  mapping_ds:{other,hashtable};
4:  spec_state_color:{orig,other};
5:  adt_observer:{orig,other};
6:  adt_subject:{orig,other};
7:  point:{orig,other};
8:  line:{orig,other};
9:  screen:{orig,other};
```

We model the dependencies among decisions as logical constraints. The following line indicates that the current design of the `line` class is based on the assumption that a hash table is used as the data structure (*mapping_ds*), the `subject` (*adt_subject*) interface is as originally agreed, and the push model is used as the notification policy (*spec_notify_policy*).

```
line = orig => adt_subject = orig &&
mapping_ds = hashtable && spec_notify_policy = push;
```

We also model test inputs (possible changes to the design) as variables and constraints. For example, adding a new observer, e.g., *new_observer_1*, to the FE design can be modeled as below:

```
new_observer_1:{orig, other};
new_observer_1 = orig => adt_observer = orig &&
spec_update_policy = simple;
```

An ACN also includes a pair-wise relation to model the *dominance* relations among design decisions, and a *clustering* relation on variables to model the fact that a design can be modularized in different ways. Our framework automatically merges a change with the design under test into one new ACN to represent the new design, computes the modularity properties of the original and new design, and generates a modularity vector as output.

2.2 Computation of Design Properties

From an ACN, we can derive a non-deterministic automaton, which we call a *design automaton* (DA), to explicitly represent the change dynamics within a design space [3, 2]. A DA captures all the possible ways in which any change to any decision in any state of a design can be compensated for by minimal perturbation, that is, changes to minimal subsets of other decisions [3, 2]. From a DA, we can also derive a *pair-wise dependence relation* (PWDR). We define two design variables to be *pair-wise dependent* if, for some design state, there is some change to the first variable for which the second must change in at least one of the minimal compensating state changes. Given an ACN, its derived DA and PWDR, our framework computes a number of modularity properties:

1. Complexity. The number of involved design dimensions, which is equal to the number of variables of the ACN.

2. Dependency Density. The coupling level of a design, reflected by the density of the PWDR pairs:

$$density = \frac{\#PWDR}{\#Variables^2} \quad (1)$$

3. Net Options Value. Another property that our framework computes is the design's Net Options Value (NOV) proposed by Baldwin and Clark [1]. The idea is that modularity provides a portfolio of options. Splitting a design into N modules increases its base value S_0 by a fraction obtained by summing the net option values (NOV_i) of the resulting options. NOV is the expected payoff of exercising a search and substitute option optimally, accounting for both the benefits and cost of exercising options:

$$V = S_0 + NOV_1 + NOV_i + ... + NOV_m$$
$$NOV_i = max_{k_i}\{\sigma_i n_i^{1/2} Q(k_i) - C_i(n_i)k_i - Z_i\}$$

For module i, $\sigma_i n_i^{1/2} Q(k_i)$ is the expected benefit to be gained by accepting the best positive-valued candidate generated by k_i independent experiments. $C_i(n_i)k_i$ is the cost to run k_i experiments as a function C_i of the module complexity n_i. $Z_i = \Sigma_{j \text{sees} i} c n_j$ is the cost of changing the modules that depend on module i. The max picks the experiment that maximizes the gain for module i. Details of the NOV model can be found in the literature [1, 17, 12]. The two most important parameters for the NOV analysis are *technical potential*, σ, and *complexity*, n.

Technical potential is the expected variance on the rate of return on an investment in producing a variant of a module implementation. Intuitively, a module that is not likely to change has low technical potential. Our framework allows the users to configure this parameter according to their own estimation. We measure the complexity of a module as the proportion of the size of the module (the number of variables) to the size of the whole ACN. For the cost of visibility, our framework automatically computes the dependents of each module, so that the users can input the costs accordingly.

2.3 Modularity Vectors

Given the design under test D and a possible change c to the design, our framework automatically merges them into one new ACN D' to represent the new design derived from applying c on D. Our testing framework produces the test output based on comparing the properties of these two ACNs: D and D', as a *modularity vector*:

$$\Delta(D' - D) = \langle \Delta size, \Delta density, modifications, \Delta nov \rangle$$

The modularity vector consists of the following dimensions: (1) $\Delta size$ models changes in the size (complexity) of the design space. Using modularization techniques incorrectly could cause class explosion, another direction of design evolution that the designer should pay attention to [6]. (2) $\Delta density$ models the changes in coupling density, assessing design coupling structure variation. (3) $modifications$ models the number of existing design decisions that have to be revisited because of *newly* introduced design dimensions, such as a new feature. This number reflects another principle of design evolution: *close to modification and open to extension* [14]. Ideally, a design should accommodate new features through extension, and avoid changing an existing part that has been debugged and proved to be correct. Our framework computes the $modification$ dimension by comparing the pair-wise dependence relation of D and D'. (4) Δnov models the changes in the options value of the design.

3. EXPERIMENTAL PROOF OF CONCEPT

Hannemann and Kiczales [8] described several possible changes to the Figure Editor system, and compared the AO [10] observer

pattern with the OO observer pattern in terms of their ability to accommodate these potential changes. We extend these changes into sequences of similar changes, and applied our modularity testing on comparing the OO and AO FE observer patterns at the design level. After that, we compare our quantitative results with the informal analysis results reported by Hannemann and Kiczales.

Hannemann and Kiczales compared the respective consequences of requiring `screen` to be a subject, and observing the positions of figure elements in addition to their colors. We answer the following questions in our experiment: (1) what are the differences between the AO and OO observer patterns in terms of their ability to accommodate additional subjects? (2) what are the differences between the AO and OO observer patterns in terms of their ability to accommodate additional states of interest that need to be observed?

Our hypothesis is that our framework should be able to quantitatively verify Hannemann and Kiczales's analysis results at the design level with a sequence of changes instead of one-step changes, and that our broader change scenarios provide additional insights in AO and OO observer patterns in terms of their ability of accommodating certain kinds of changes.

Designs Under Test. We consider two designs as the software artifacts under test: the Figure Editor observer patterns designed using object-oriented and aspect-oriented paradigms, respectively. We formalize these two designs as ACNs, each representing the major design dimensions and their dependency relations. For example, in the AO design, an abstract aspect is employed to encapsulate such decisions as what data structure is used to store the mapping between the observer and the subjects. This abstract aspect can be extended with other aspects, serving as an interface.

Test Inputs. We use two sequences of changes as test inputs: (1) *adding 1 to n subjects*; (2) *adding 1 to n new states of interest.* Each sequence of changes includes n changes (each adding one subject or state at a time) being applied one after another accumulatively to the design under test. For example, according to Hannemann and Kiczales's paper, each new state of interest is handled by a new protocol inherited from the abstract protocol. We model this change as:

```
state_n_con_protocol:{orig,other};
state_n_con_protocol = orig =>
abstract_protocol = orig && line = orig &&
point = orig && screen = orig &&
spec_update_policy = simple;
```

Testing Results. To decide which design (OO or AO) can provide better modularity, i.e., can better accommodate envisioned changes, we compare the modularity vectors produced by our testing framework. From Tables 1 and 2, we first observe that at the beginning, AO is better than OO: (1) the lower dependency density indicates fewer couplings; (2) the higher NOV value means that there are more independent modules free to be substituted with better versions.

(1) *What are the differences between the AO and OO observer patterns in terms of their ability to accommodate additional subjects?* In our experiment, we test five changes, each of which accumulatively adds one subject to the design under test. Table 1 shows the test input-output pairs for the OO and AO designs. We observe that when adding new subjects, the coupling level of the OO design increases while the coupling level of the AO design decreases. Although the NOV values of both designs increase with new subjects, the NOV value of the AO design increases more than that of the OO design. We also observe that all the AO designs have lower density and a higher NOV value than that of OO designs, and that this trend continues if more subjects are added. By comparing modularity vectors, our framework quantitatively shows that the AO design is better than the OO design in term of the ability of accommodating more subjects.

(2) *What are the differences between the AO and OO observer patterns in terms of their ability to accommodate additional states of interest that need to be observed?* In our experiment, we test five changes, each of which accumulatively adds one state to the design under test. From Table 2, we observe that (1) the AO design follows the open to extension and close to modification principle: no existing design dimensions will be affected by these changes. However, (2) although the original AO design has lower density than the original OO design, when adding new states of interest, the coupling level of the OO design decreases dramatically while the density of the AO design keeps increasing. (3) Although the NOV values of both designs increase with new states, the NOV value of the OO design increases more than that of the AO design. We also observe that if fewer than 2 states are added, the AO design has lower coupling and a higher NOV value. However, if more states are to be added, the density of the AO design will increase dramatically and the NOV values get lower. The reason is that when a concrete aspect protocol is added for one state observation, it will depend on the subjects. When more protocols are added, the subjects will have more and more dependents, and become harder and harder to change, which causes the NOV values to decrease.

Summary. Hannemann and Kiczales concluded that the AO design is superior in terms of both changing the role of `screen`, making it both a subject and an observer, and observing positions additionally. Our analysis showed that in terms of these one-step changes, their conclusion is correct. But if there are more states to be observed, the AO design is not better in the long run using the current design of creating a new protocol for each new state.

4. RELATED WORK

Baldwin and Clark's *Net Option Value* (NOV) [1] analysis provides a general way to statically and quantitatively assess design modularity based on design structure matrices (DSM) modeling. Both Sullivan et al. [17] and Lopes et al. [12] applied this analysis to software design comparison and evaluation. Instead of using one number to assess design modularity, our testing framework allows the designer to test design modularity from multiple dimensions, making tradeoffs among these dimensions explicit. Garcia et al. [7] quantitatively compared AO and OO solutions for design patterns using a suite of metrics, such as coupling, cohesion, and size. Our work is different in that our analysis works at the design level, tests design modularity in terms of changeability, considers a sequence of possible changes, and integrates net option value analysis.

Various previous approaches on testing or analyzing software designs focus on functional correctness of software designs. For example, Jackson et al. [9] developed the Alloy Analyzer to analyze a software design written in the the Alloy modeling language against user-specified properties. Dinh-Trong et al. [5] developed a tool for generating test inputs for UML design models such as UML class and sequence diagrams and checked the execution of the generated tests against common properties or user-specified properties. Recently modularity on software designs has been exploited to support modular verification [11, 4] of the designs against functional correctness. Different from these previous approaches of testing or analyzing software designs for functional correctness, our framework focuses on testing modularity of software designs.

5. FUTURE WORK AND CONCLUSION

In order to assess a software design's modularity properties rigorously against its ability to accommodate changes, we have devel-

Table 1: Comparison of test input/output pairs of AO vs. OO design in terms of adding subjects

Design under test						Test input	Test output							
size		density		NOV			Δsize		Δdensity		modification		ΔNOV	
OO	AO	OO	AO	OO	AO		OO	AO	OO	AO	OO	AO	OO	AO
9	9	17.28%	9.88%	3.17	3.84	0: Screen as an subject	0	0	4.94%	**0%**	**0**	1	-0.45	**0**
9	9	22.22%	9.88%	2.72	3.84	1: add 1 new subject	1	1	5.72%	**-0.88%**	**0**	1	-0.30	**0.43**
10	10	23.00%	9.00%	2.87	4.27	2: add 2 new subjects	2	2	5.86%	**-1.61%**	**0**	1	-0.06	**0.83**
11	11	23.14%	8.26%	3.11	4.67	3: add 3 new subjects	3	3	5.63%	**-2.24%**	**0**	1	0.22	**1.19**
12	12	22.92%	7.64%	3.39	5.03	4: add 4 new subjects	4	4	5.20%	**-2.78%**	**0**	1	0.63	**1.54**

Table 2: Comparison of test input/output pairs of AO vs. OO design in terms of adding states

Design under test						Test input	Test output							
size		density		NOV			Δsize		Δdensity		modification		ΔNOV	
OO	AO	OO	AO	OO	AO		OO	AO	OO	AO	OO	AO	OO	AO
9	9	17.28%	9.88%	3.17	3.84	0: Positions as a new state	1	1	**-1.28%**	3.12%	2	**0**	**0.40**	0.03
10	10	16.00%	13.00%	3.57	3.87	1: add 1 new state	2	2	**-2.41%**	5.00%	2	**0**	**0.77**	0.10
11	11	14.88%	14.88%	3.94	3.94	2: add 2 new states	3	3	**-3.40%**	6.10%	2	**0**	**1.11**	0.20
12	12	13.89%	15.97%	4.28	4.04	3: add 3 new states	4	4	**-4.27%**	6.69%	2	**0**	**1.44**	0.31
13	13	13.02%	16.57%	4.61	4.15	4: add 4 new states	5	5	**-5.04%**	6.96%	2	**0**	**1.75**	0.46

oped a novel modularity testing framework. In this framework, we model software designs and potential changes uniformly using augmented constraint networks, independent of language paradigms and modularization techniques. We define *modularity vectors* to quantitatively reflect a number of informal design principles, and to integrate the net option value analysis. Using this framework, we analyzed the object-oriented observer pattern versus aspect-oriented observer pattern in terms of their ability to accommodate a sequence of envisioned changes. The result shows that our framework quantitatively and formally verified previously informal analysis results, and provides additional insights.

Sometimes designers may not be certain on what kinds of potential changes that can happen in the future. In this situation, we can enumerate possible changes to *cover* each design dimension (coverage is defined based on whether a change occurs on the design dimension) and then analyze the overall modularity vectors for these changes without requiring the designer to come up with possible changes. The analogy in traditional software testing is to enumerate possible test inputs to cover each branch where branches are analogous to design dimensions. In future work, we plan to define a set of coverage criteria for a design in assessing how well the generated changes cover the design space.

6. REFERENCES

[1] C. Y. Baldwin and K. B. Clark. *Design Rules, Vol. 1: The Power of Modularity*. The MIT Press, 2000.

[2] Y. Cai. *Modularity in Design: Formal Modeling and Automated Analysis*. PhD thesis, University of Virginia, Aug. 2006.

[3] Y. Cai and K. Sullivan. Simon: A tool for logical design space modeling and analysis. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 329–332, Nov 2005.

[4] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Trans. Softw. Eng.*, 30(6):388–402, 2004.

[5] T. T. Dinh-Trong, S. Ghosh, and R. B. France. A systematic approach to generate inputs to test UML design models. In *Proceedings of the 17th International Symposium on Software Reliability Engineering*, pages 95–104, 2006.

[6] R. J. Erich Gamma, Richard Helm and J. Vlissides. *Design Patterns: Elements of Resuable Object-Oriented Software*. ADDISON-WESLEY, Nov 2000.

[7] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa. Modularizing design patterns with aspects: a quantitative study. In *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, pages 3–14, 2005.

[8] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 161–173, 2002.

[9] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Cambridge, MA, 2006.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 220–242, 1997.

[11] H. C. Li, K. Fisler, and S. Krishnamurthi. The influence of software module systems on modular verification. In *Proceedings of the 9th International SPIN Workshop on Model Checking of Software*, pages 60–78, 2002.

[12] C. V. Lopes and S. K. Bajracharya. An analysis of modularity in aspect oriented design. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pages 15–26, 2005.

[13] A. Mackworth. Consistency in networks of relations. In *Artificial Intelligence, 8*, pages 99–118, 1977.

[14] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[15] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–8, Dec. 1972.

[16] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–39, 1974.

[17] K. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. In *ESEC/FSE '01*, pages 99–108, Sept 2001.