A Decision-Tree Model for Software Evolution Analysis

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

ABSTRACT

A modularization technique, or a refactoring method, benefits a design if the potential changes to the design can be well encapsulated by the technique. In general, questions in software evolution, such as which modularization technique is better and whether it is worthwhile to refactor, should be evaluated against potential changes. In this paper, we present a decision-tree-based framework to generally assess design modularization in terms of its changeability. In this framework, we formalize design evolution questions as decision problems, model software designs and potential changes using augmented constraint networks (ACNs), and represent design modular structure before and after envisioned changes using design structure matrices (DSMs) derived from ACNs. We formalize change impacts using an *evolution vector* to precisely capture well-known informal design principles. For the purposes of illustration and preliminary evaluation, we use this model to compare the aspect-oriented and object-oriented observer pattern in terms of their ability to accommodate envisioned changes. The results confirm previous results, but in formal and quantitative ways. We also show that this approach can be generalized to real designs by applying our approach to the partial refactoring decision-making problem of the Galileo fault tree analysis tool.

1. INTRODUCTION

People have long recognized that evolvability, achieved most fundamentally by appropriate modularity in design, can have enormous technical, organizational and ultimately economic value. However, as a discipline, we still lack the basic science needed to analyze fundamental design decisions such as whether it is worthwhile to refactor existing system, or how to best accommodate envisioned changes.

Each modularization technique, such as aspect-programming techniques or a design pattern, provides one way to let some part of a system change independently of all other parts. A modularization technique, or a refactoring method, benefits a design when the potential changes to the design can be well encapsulated by the technique [20, 24, 14, 5]. In this paper, we present a decision-tree-based framework to generally assess design modularization in terms of its changeability.

Analyzing design modularity in general requires a design modeling technique independent of particular modularity techniques and language paradigms. We have developed a framework to formally model software designs using augmented constraint networks (ACNs) [6, 4], in which design dimensions and environment conditions are uniformly modeled as variables, possible choices as values of variables, and the relations among decisions as logical constraints. The ACN modeling formalizes the key notions of Baldwin and Clark's design rule theory and Parnas's information hiding criteria [2, 20, 4]. The supporting prototype tool, Simon, supports basic design impact analysis and the automatic derivation of *design structure matrices* (DSMs) [6, 7, 4].

In our framework, we first use ACNs to formally model both software designs and potential changes, and represent design modular structure using DSMs automatically derived from ACNs. Second, we formalize design evolution questions as decision problems, modeling the decision making procedure as a *design tree* (DT). Each node of a DT models a evolution decision, such as the choices of design patterns, a refactoring mechanism, or a potential change. Each decision is associated with an ACN modeling the design resulting from the decision. Finally, the user can compare the design modular structure before and after envisioned changes using design structure matrices (DSMs) and ACNs. We formalize the comparison using a *evolution vector* to capture well-known but informal design principles, such as maximizing cohesion, minimize coupling, open to extension, close to modification [21, 12].

People have been analyzing design evolvability and changeability in qualitative, intuitive, and heuristic ways. For example, Hannemann and Kiczales used a well known and widely-used Figure Editor (FE) [15, 14, 11] example to compare the evolvability and modularity property of using aspect-oriented (AO) versus object-oriented (OO) paradigm to implement design patterns. They show the actual code implementing these choices as the evidence of their analysis. However, designers frequently face questions of the like before coding. For the purpose of illustration and preliminarily evaluation, we model the comparison of AO observer pattern vs. OO observer pattern using our evolution model against

potential changes mentioned in their paper, and compare our quantitative results with their informal analysis.

To show that our approach can be generalized to real designs, we model and analyze part of the Galileo dynamic fault tree analysis tool, developed at the University of Virginia for production use at NASA [25, 9]. The Galileo designers once faced a situation when they had to make a decision about how to restructure part of the system. They reached a decision based on discussions and arguments, rather than rigorous analysis. Modeling and analyzing this historical scenario using our decision tree model, the designers are now able to compare different decisions comprehensively and justify their decision quantitatively.

Through these case studies, we show that (1) our framework is general enough to account for the OO and AO modularity in uniform, declarative terms; (2) our framework automates Hannemann and Kiczales's evolvability and modularity analysis precisely; and (4) the result is both generalizable and scalable to real designs.

The rest of this paper is organized as follows. Section 2 uses the figure editor example to illustrate how to generally represent software designs using ACN and DSM models, independent of modularization techniques and language paradigms. Section 3 presents our evolution model and preliminary evaluation results. Section 4 presents our case study on Galileo. Section 5 discusses related work. Section 6 concludes.

2. DESIGN REPRESENTATIONS

In this section, we use a Figure Editor (FE) example [16, 14] to illustrate how to use design structure matrices (DSMs) and augmented constraint networks (ACNs) to generally model software designs and potential changes, which provides the foundation for our changeability analysis framework. The Figure Editor is a tool for editing drawings comprising points and lines (figure elements), where a screen displays each figure element, always reflecting the figure elements' current states.

2.1 Design Structure Matrices

Design Structure Matrix (DSM) modeling originated with the work of Steward dating to the 1960s [22], and has been further developed and applied in the design, analysis and management of many large-scale engineering systems by Eppinger [10] and others. DSMs are the primary representations at the heart of Baldwin and Clark's developing theory of the economics of modularity [2].

DSMs present in a graphical form the pair-wise dependence structure of designs. The upper left DSM shown in Figure 2 models the figure editor design using object-oriented observer pattern (generated by Simon). The rows and columns of a DSM are labeled with design variables, representing dimensions for which the designers must make design decisions. A marked cell indicates that the decision of the dimension on the row depends on the decision of the dimension on the column. The cell in row 6, column 1, indicates that how the *Point* should be designed depends on the notification policy in use.

DSM modeling is capable to represent a wide range of design decisions, to model *dominance relations*, a key property of Baldwin and Clark's notion of design rule, by asymmetric dependencies, and to represent multiple modularization methods by reordering the columns and rows of a matrix. Given these advantages, Sullivan et al. [23] showed that Baldwin and Clark's DSM can be extended with environment parameters, and thus precisely account for Parnas's information hiding criterion.

However, DSM modeling does not appear to be expressive enough to support precise design analysis or a rigorously formal theory of coupling in design. First, we have found that building such design models manually is error-prone and time-consuming. Our recent work [6] has shown errors in published DSMs. Many of the errors are due to the difficulty of seeing transitive relations among dependencies; or to the lack of any precise definition of dependence. Second, a DSM only represents design dimensions, but not concrete choices within each dimension nor the semantics of the constraints that relate decisions across dimensions. For example, Gamma et al. [11] mentioned that possible choices for the *notification policy* could be either *push* or *pull*, each having different consequences. DSMs do not explicitly express these choices, nor do they support the analysis of their consequences. Third, there are usually multiple ways to accommodate a change, but a DSM model does not reveal them, and the exact meaning of a mark becomes ambiguous.

2.2 Augmented Constraint Network

To address these problems, our recent work [6] presents the *Augmented Constraint Network* (ACN) as a formal design representation better subject to automated analysis of the design evolvability and economic-related properties. A DSM can be automatically generated from an ACN. As a result, our ACN modeling has the advantages of DSM modeling and overcomes the shortcomings. The core of an ACN is a finite-domain *constraint network* (CNs) [19], which consists of a set of *design variables* modeling design dimension or relevant environmental condition, 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. For example, we can model the *notification_policy* as the following scalar design variables: $spec\_notify\_policy(push, pull)$.

A design decision or environmental condition is represented by a binding of a *value* from a *domain* to a variable. We model dependencies among decisions as logical constraints. Figure 1 shows a constraint network modeling object-oriented observer pattern. Line 13 indicates that the current *adt_subject* design is based on the assumption that a hash table is used as the data structure (*mapping_ds*) , the `Observer` interface is as originally agreed (*adt_observer*), and the push model is used as the notification policy (*spec_notify_policy*). 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.

Hannemann and Kiczales [15] mentioned several changes of the observer pattern, for example, what if the client requires the `Screen` to be both a subject and an observer? We can model such a change as a constraint too. Making `Screen` both a subject and an observer just requires the

```
1: spec_notify_policy:{push,pull};
2: spec_update_policy:{orig,other};
3: mapping_ds:{hash,other};
4: color_policy_observing:{orig,other};
5: adt_observer:{orig,other};
6: adt_subject:{orig,other};
7: point:{orig,other};
8: line:{orig,other};
9: screen:{orig,other};
10: line = orig => adt_subject = orig &&
  color_policy_observing = orig;
11: screen = orig => adt_observer = orig;
12: screen = orig => spec_update_policy = orig;
13: adt_subject = orig => mapping_ds = hash &&
  adt_observer = orig &&
  spec_notify_policy = push;
14: point = orig => adt_subject = orig &&
  color_policy_observing = orig;
```

Figure 1: The FE OO Observer Pattern

`Screen` class extend the abstract subject interface, and respect color observing policy, such as invoking notification function whenever the color changed:

```
screen = orig => adt_subject = orig &&
color_policy_observing = orig;
```

We augment a CN with a pair-wise relation to model the *dominance* relations among design decisions. For example, agreed interfaces often dominate subsequent implementations. Another instance is that environment conditions are usually out of the designer's control. For example,
`(subject, spec_notify_policy)`
is a member of the *dominance* relation, modeling that the decision on notification policy is dominating. Another augmentation is a *clustering* relation on variables to model the fact that a design can be modularized in different ways.

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 [6, 7, 4]. A DA captures all of 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, enabling basic *design impact analysis* (DIA) that have fully automated and quantified Parnas's changeability analysis [6, 4].

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. From a derived PWDR and a selected clustering method of the ACN, a DSM can be automatically generated by our tool, Simon. As a result, we can in principle apply all the analysis capabilities developed for DSMs to our much more complete and precise models.

Although the basic *design impact analysis* is sufficient to analyze the impact of changes in design decisions within a design space determined by an ACN, it is not adequate to analyze the impact of design decisions that could change the structure of the design space, for example, by introducing new design dimensions. The decisions to refactor the existing system and to apply a new design pattern are examples. In the next subsection, we model such design decisions using a decision-tree based framework to generally assess design modularity and evolvability.

3. SOFTWARE EVOLUTION MODEL

Hannemann and Kiczales [15] mentioned several possible changes in Figure Editor observer pattern design, and compared the AO observer pattern with the OO observer pattern in terms of their ability to accommodate these changes. To illustrate our approach, we formalize the comparison problem using our decision-tree-based framework, and quantify the comparision using *evolution vectors*. To evaluate our apporach, we compare our quantitative results with their informal analysis results.

3.1 Decision Tree Modeling

Hannemann and Kiczales's analysis focused on answering the following changeability questions: (1) what are the consequences if the client changes the *role assignment*, requiring the `Screen` to be both a subject and an observer? (2) In the original design, the color of subject figure elements is the only state of interest that needs to be observed. What if the positions of the figure elements also need to be observed?

We formulate their analysis as decision problems modeled by a decision tree shown in Figure 2. The node V_0 is the starting point. Square nodes represent design decisions, and round nodes represent the resulting design of a given decision. For example, square node *AO* represents the decision to use AO paradigm. Node D_{AO} represents the corresponding AO design.

Figure 2 models a decision-making procedure: we first represent the choice of using AO or OO paradigm as two decisions leading to two designs, D_{AO} and D_{OO}. We model each design as an ACN. Figure 1 and Figure 3 show the constraint networks of the D_{OO} and D_{AO} respectively. Note that we represent object-oriented design and aspect-oriented design in an uniformed way. In the aspect 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. We model the interface part of the abstract aspect as `abstract_protocol`, as shown in line 8, Figure 3, and model the decisions it encapsulates as `abstract_protocol_impl` (line 9).

After that, we view the envisioned two changes, the new *role assignment* and the additional *position observing*, as subsequent decisions, and model these decisions as variables and model their relations as logical constraints. As an example, Figure 4 shows the constraint network of using AO design to accommodate the new position observing policy. This partial constraint network will be combined with the original one as shown in Figure 3 to form a new design, D_{AO_pos}.

Making the two changes in the OO and AO designs respectively leads to four designs, D_{OO_role}, D_{OO_pos}, D_{AO_role}, and D_{AO_pos}, each represented by a new ACN. We derive the

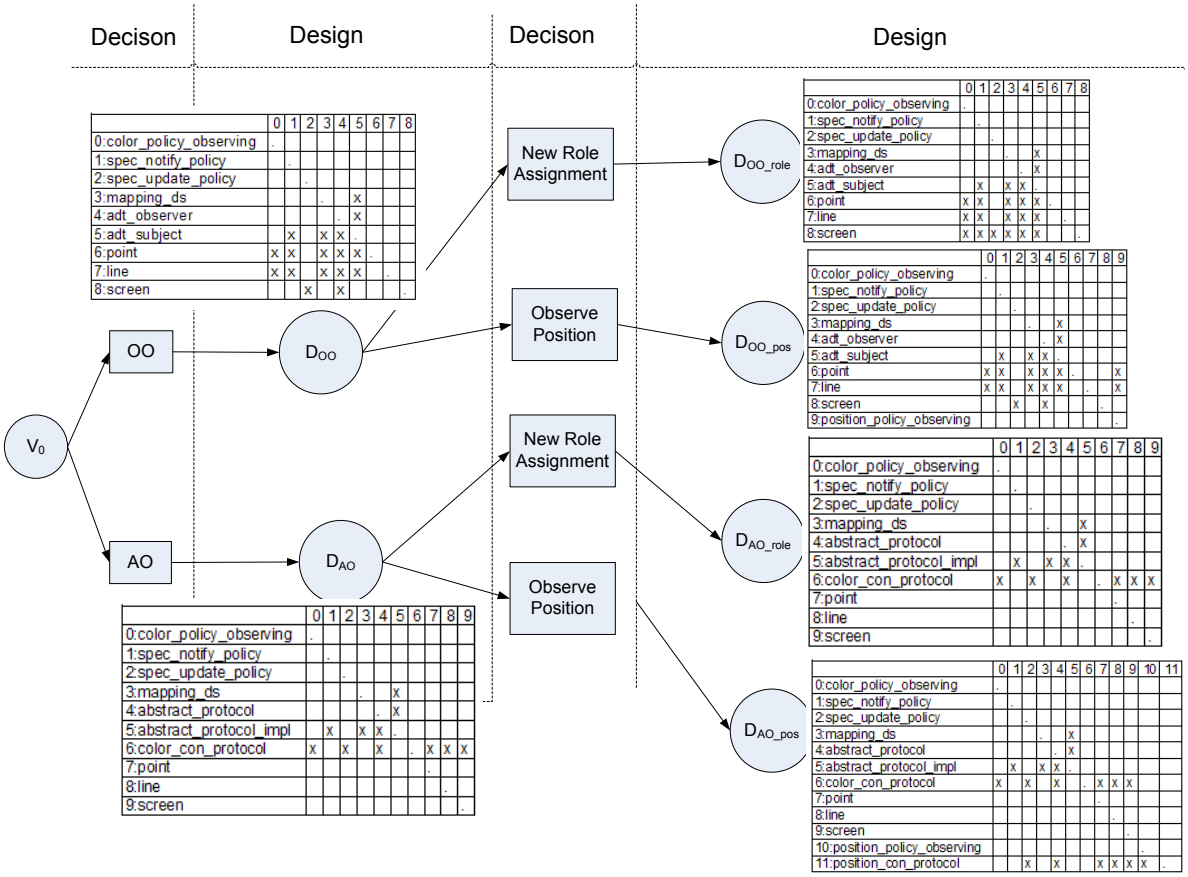

Figure 2: Software Evolution Model

```
1: spec_notify_policy:{push,pull};
2: spec_update_policy:{orig,other};
3: mapping_ds:{hash,other};
4: color_policy_observing:{orig,other};
5: point:{orig,other};
6: line:{orig,other};
7: screen:{orig,other};
8: abstract_protocol:{orig,other};
9: abstract_protocol_impl:{orig,other};
10: color_con_protocol:{orig,other};
11 abstract_protocol_impl = orig => mapping_ds = hash
&& abstract_protocol = orig
&& spec_notify_policy = push;
12: color_con_protocol = orig =>
color_policy_observing = orig;
13: color_con_protocol = orig =>
abstract_protocol = orig
&& line = orig && point = orig && screen = orig
&& spec_update_policy = orig;
```

Figure 3: The FE AO Observer Pattern

```
1: position_con_protocol:{orig,other};
2: position_policy_observing:{orig,other};
3: position_con_protocol = orig =>
position_policy_observing = orig;
4: position_con_protocol = orig =>
abstract_protocol = orig && line = orig &&
point = orig && screen = orig &&
spec_update_policy = orig;
```

Figure 4: A Change: Position Observing

DSMs of all six ACNs, as associated each design node. For example, the D_{OO} DSM is derived from the ACN in Figure 1. To decide which paradigm can better accommodate envisioned changes, we just need to compare the differences between the designs with and without these changes. To facilitate the comparison, we first formalize the differences between two designs using an *evolution vector* to capture several well-know but informal design principles.

3.2 Evolution Vector

Given an ACN and the derived *pair-wise dependence relations* (PWDR), the coupling level of a design can be reflected by the density of the pairs:

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

For example, the density of D_{OO} is 21%. We define the differences of two designs, D and D', as an *evolution vector*: $\Delta(D' - D) = \langle \Delta density, \Delta modifications, \Delta size \rangle$,
which consists of the following dimensions, each capturing an informal design principle:

- $\Delta density$ models the changes in coupling density, assessing design coupling structure variation. Minimizing coupling is an important, well-known, but informal design principle [21]. A negative $\Delta density$ indicates decreased coupling among design decisions, which is favorable.

- $\Delta modifications$ 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* [12]. Ideally, a design should accommodate new features through extension, and avoid changing existing part that has been debugged and proved to be correct. This number only applies when new dimensions are added to the design. To determine the effects of changing the decision on an exsiting dimension, that is, changing the value of a variable, the basic *design impact analysis* suffices in that it computes in detail what are the possible changes to which other variables [6].

- $\Delta size$ models changes in the size of the design space. More design dimensions indicates more complexity. Using modularization techniques incorrectly could cause class explosion, another direction of design evolution the designer should pay attention to [11].

We have shown that the ACN and DSM modeling can precisely capture Parnas's information hiding criteria, against with each design can be evaluated [23, 6, 7, 4]. We have also shown that using the design impact analysis function of Simon, we can analyze the respective consequences of changing common design decisions for two design alternatives, such as the different impact of changing the notification policy from `push` to `pull` in AO and OO designs respectively. Simon shows that in the OO design, three variables have to be revisited, while in the AO design, only one variable should be revisited. Counting the number of variables affected by a change in a decision is clearly insufficient to determine costs of change, but identifying what must change is a critical step.

The information hiding and design impact analysis supported by Simon work within a single ACN model. By contrast, the evolution vector reflects structural changes between two designs, and the vector summarizes the results of performing design impact analysis within each design respectively.

In addition, the evolution vector is extensible. For example, we could add a new dimension, Δnov, to calculate Baldwin and Clark's *net option value* (NOV) based on DSM modeling [2]. The NOV analysis involves estimating such parameters as technical potentials, which is out of the scope of this paper, and we only consider evolution vectors with the above three dimensions.

3.3 Analysis Results

Given the decision tree model and the evolution vector, we can now quantitatively assess the OO vs. AO observer pattern modularity against the envisioned changes:

(1) *What is the difference of using AO vs. OO to design an observer pattern?*

To analyze this problem, we need to compute $\Delta(D_{AO} - D_{OO})$. D_{OO} has 9 variables, D_{AO} has 10 variables, and $\Delta size = 1$. D_{OO} has 17 coupling pairs, and $density = 21\%$, D_{AO} has 11 pairs, and its $density = 11\%$. As a result, $\Delta density = -10\%$. Since we are not modeling how to change an OO design to an AO design, the $\Delta modificaiton$ is not applicable. As a result, we get:
$\Delta(D_{AO} - D_{OO}) = \langle \Delta density = -10\%, N/A, \Delta size = 1 \rangle$,
showing that the AO design lowers the coupling level but slightly increases the complexity.

From the DSMs associated with D_{AO} and D_{OO}, we observe that the decisions on the notification and update policies no longer influence concrete subjects, *Point* and *Line*. Instead, only the abstract and concrete protocols depend on these policies, indicating the localization of crosscutting decisions.

(2) *What is the impact of changing the* role assignment *so that the* Screen *is both a subject and an observer?*

We first observe that no new dimension is introduced by this change, and only the constraints among the variables are changed. As a result, the $\Delta modification$ dimension is not applicable. To analyze the different impact of change on the OO and AO design respectively, we first the following evolution vectors:

- The change impact on the OO design:
  $\Delta(D_{OO\_role} - D_{OO}) =$
  $\langle \Delta density = 5\%, N/A, \Delta size = 0 \rangle$,
  meaning that the level of coupling increases.

- The change impact on the AO design:
  $\Delta(D_{AO\_role} - D_{AO}) =$
  $\langle \Delta density = 0, N/A, \Delta size = 0 \rangle$,
  which indicates that the coupling density remains unchanged after accommodating this new feature.

Comparing these vectors quantitatively proves that the AO design is better than the OO design in term of the ability of accommodating this particular change. From the DSMs, we can also observe that the D_{AO_role} DSM is exactly the same as the DSM of D_{AO}, which means that the AO design localizes this change completely without incurring any additional dependencies or new dimensions. The result shows that the AO paradigm has obvious advantages over the OO paradigm in terms of accommodating this particular change.

(3) *What if the* observing policy *changed so that the positions of the figure elements should also be observed in addition to the colors?*

To analyze this problem, we compute the following vectors:

- The change impacts on the OO design:
  $\Delta(D_{OO\_pos} - D_{OO}) =$
  $\langle \Delta density = -2\%, \Delta modifications = 2, \Delta size = 1 \rangle$.
  From this vector, we observe that although the level of coupling is decreased, two existing decisions have to be revisited.
- The change impact on the AO design:
  $\Delta(D_{AO\_pos} - D_{AO}) =$
  $\langle \Delta density = 1\%, \Delta modifications = 0, \Delta size = 2 \rangle$.

We observe that although the AO design incurs more dependences to accommodate this change, the existing system is not affected because the $\Delta modification = 0$. It indicates that the design can be extended to accommodate this particular change without affecting existing decision. On the other hand, we need one more variable `position_con_protocol` to model the new aspect handling position observation, slightly increasing design complexity.

At this point, if there are additional possible changes, we can extend the decision tree for further analysis. For example, if more figure elements are going to be added, taking the roles of subjects, such as circles and triangles, the AO paradigm will be better since it can localize the task of role assignment. On the other hand, if the figure elements are fixed, but the number of states that need to be observed keeps increasing, using AO design requires adding more aspects for each observable dimension, which will increase the complexity of the whole design.

In summary, using the decision tree model and evolution vectors, we have quantitatively captured Hannemann and Kiczales's qualitative analysis, revealing how these high-level design decisions impact the design coupling structure visually and precisely.

4. GALILEO

To show that our approach can be generalized to real systems, we apply it to Galileo, a dynamic fault tree analysis tool [25, 9]. Galileo has about 35,000 lines of C++ code excluding library and generated code. The design and implementation of this system was independent of the work presented in this paper. It has evolved continually over eight years, with hundreds of files and many student developers.

During maintenance and feature enhancement stages, we found several problematic issues. In particular, we found that it is hard to justify different design error-handling refactoring proposals. We modeled these historical situations using the decision tree model, compute the evolution vectors for each proposed refactoring method with envisioned future changes, and found that our analysis provides quantitative results that are consistent with our earlier refactoring decisions.

Error handling was suggested to be refactored to be more consistent and descriptive. Two dimensions involved in error handling were the support for multiple error types (e.g., syntax error and semantics error), and the support for multiple views (Word97, Visio5). According to different types of errors, the error handling module should mark the views where an error happens, jump to the error point, give messages, and clear the marks once the error is corrected. We call these actions a marking sequence, modeled by `MarkSequence`.

Four refactoring mechanisms were proposed in relation to the addition of sophisticated error handling to Galileo. The designer faced the problem of choosing the best one. We modeled this situation as nodes $D1$, $D2$, $D3$, $D4$ in the decision tree as shown in Figure 5.

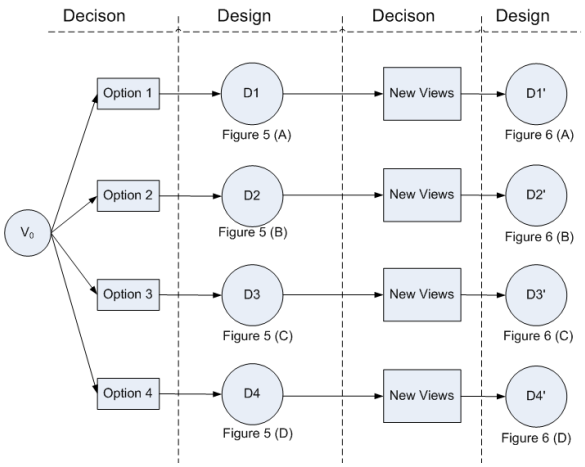

Figure 5: Select a refactoring method

The first option requires that each error object knows in which view an error happens, and implements the marking sequence. Figure 6 shows the constraint network modeling the constraints among the view types, error types, and the mark sequence in option 1. The DSM generated by Simon for option 1 is shown in Figure 7 (A) The second option is symmetric to the first one, requiring that each view knows what type of error happened, and that it then implements the marking sequence. Figure 7 (B) shows the DSM for option 2.

At this point, a marker class, modeled by `markers`, was proposed to take the responsibility of implementing a marking sequence for each combination of error and view types. Given the marker classes, the third and fourth options were proposed. The major difference of the third and fourth options was in who deciding which error happened in which view and in invoking the corresponding marker object. The third option required each error objects to take this responsibility and the fourth option demanded a new class, `ErrorToMark`, to do the job. We similarly modeled these

```
1: word_view:{orig,other};
2: visio_view:{orig,other};
3: MarkSequence:{orig,other};
4: syntax_errors:{orig,other};
5: semantics_errors:{orig,other};
6: word_view = orig => syntax_errors = orig &&
semantics_errors = orig && MarkSequence = orig;
7: visio_view = orig => syntax_errors = orig &&
semantics_errors = orig && MarkSequence = orig;
```

Figure 6: Error Handling Refactoring Option 1

options as augmented constraint networks, and generated their DSMs using Simon, as shown in Figure 7 (C) and (D).

By comparison, we can tell that options 1 and 2 are simpler in the sense that they involve fewer design dimensions. However, although the number of dependences and number of variables are fewer in the option 1 and 2, the density of the dependencies in these two DSMs are higher: of the 25 cells, there are 12 cells (48%) marked as dependencies. The dependence density is 44% for option 3, and is 26% for the DSM of option 4. Option 4 is the best in terms of coupling structure.

We now evaluate which option is best against possible future changes. One envisioned change is to add new views to the system, for example, an Excel view and an XML view, modeled by two new variables, `excel_view` and `xml_view`. We model the decision to add these new views as new decisions, shown as node D1', D2', D3', and D4'. For each decision We develop an ACN, and generate their DSMs as show in Figure 8.

To quantitatively analyze these decisions, we compute the following evolution vectors:

- $\Delta(D1' - D1) =< \Delta density = 2\%, \Delta modifications = 3, \Delta size = 2 >$;
- $\Delta(D2' - D2) =< \Delta density = -15\%, \Delta modifications = 2, \Delta size = 2 >$;
- $\Delta(D3' - D3) =< \Delta density = -36\%, \Delta modifications = 3, \Delta size = 6 >$;
- $\Delta(D4' - D4) =< \Delta density = -30\%, \Delta modifications = 2, \Delta size = 6 >$;

We observed that the design spaces are expanded similarly in all cases. Although the sizes of option 1 and option 2 increased less under the change, the design decisions are highly coupled. Given the change, the coupling density increased in option 1. The coupling density of option 3 and option 4 are dramatically decreased after the change, meaning that the newly added dimensions accommodated most of the changes.

Comparing option 3 and option 4, although the density decrement of option 4 is a little less than that of option 3, the design of option 4 (18%) has much less dependence density than option 3 (28%). In addition, in the design of option 4, fewer existing decisions will be affected.

In real implementation, the major difference is that when new views are added, options 1, 2, and 3 require changing more places than option 4 does. For example, according to Figure 8, for the design using option 3, `syntax_errors` and `synmantics_errors` are going to be changed. The option 4 design requires only change to `ErrorToMark`. This analysis quantitatively validates the choice that we actually made: to use the fourth option.

In summary, our experiments support the claim that our approach can be generalized to real systems, and that our framework is expressive enough to capture varieties of design phenomena uniformly and analyze these problems automatically, precisely and quantitatively.

5. RELATED WORK

Baldwin and Clark's *Net Option Value* (NOV) [2] analysis provides a general way to statically and quantitatively assess design modularity based on DSM modeling. Both Sullivan et al. [23] and Lopes et al [18] applied this method to software design comparison and evaluation. The NOV analysis also takes into account the size of the design (complexity), and the ripple effects. The challenge for NOV analysis is the necessity to estimate technical potential of each module. Instead of using one number to assess design modularity, our approach allows the designer to comprehensively evaluate design modularity and changeability from multiple angles, making the trade-offs among between these dimensions explicitly.

Software design space, feature modeling and variability modeling in product family modeling has been widely studied. Feature modeling supports automatic program variations, which have also been widely studied in Batory's work on generic programming [3], Goguen's work on parameterized programming [13] and Czarnecki work on generative programming [8]. While their purpose is to synthesize complex software systems from libraries of reusable components, our purpose is to rigorously support modularity analysis and decision-making. Our approach is more general in that we not only model and analyze features, which are one kind of design decisions, but also broader decisions such as refactoring options, design patterns, and aspects. We aim to analyze the design modular structures and their implications, while they aim to analyze feature properties.

Similar to our design space modeling, Lane [17] models the structure of software systems as design spaces by identifying the key functional choices, and classifying the alternatives available for each choice. Their notions of rules, similar to our constraints, are formulated to relate choices within a design space. Our approach is more general in that we model broader decision-making phenomena and their impacts, such as a decision to choose a design pattern or a modularization technique. Traditional impact analysis research focuses on change issues at program level, as summarized in [1], while our approach works at abstract design level.

6. CONCLUSION

In order to assess software modularity uniformly against its ability to accommodate changes, we presented a decision-tree-based assessment framework to facilitate design changeability analysis. In this framework, we model software designs and potential changes uniformly using augmented constraint networks, independent of language paradigm and modularization techniques. We model design modular structure using derived design structure matrices, and define *evolution vectors* to quantitatively reflect a number of informal design principles. Using this framework, we analyzed the object-oriented observer pattern versus aspect-oriented observer pattern in terms of their ability to accommodate envisioned changes, and similarly compared four refactoring methods for Galileo error handling. The result shows that our model quantitatively and formally verified previously

	0	1	2	3	4
0:word_view	.		x	x	x
1:visio_view		.	x	x	x
2:syntax_errors	x	x	.		
3:semantics_errors	x	x		.	
4:MarkSequence	x	x			.

(A) Error Handling Option 1

	0	1	2	3	4
0:word_view	.		x	x	
1:visio_view		.	x	x	
2:syntax_errors	x	x	.		x
3:semantics_error	x	x		.	x
4:MarkSequence			x	x	.

(B) Error Handling Option 2

	0	1	2	3	4	5	6	7	8
0:word_views	.		x	x					
1:visio_views		.	x	x					
2:syntax_errors	x	x	.		x	x	x	x	x
3:semantics_errors	x	x		.	x	x	x	x	x
4:syntax_word_markers			x	x	.				x
5:syntax_visio_markers			x	x		.			x
6:semantics_word_marker			x	x			.		x
7:semantics_visio_marker			x	x				.	x
8:MarkSequence			x	x	x	x	x	x	.

(C) Error Handling Option 3

	0	1	2	3	4	5	6	7	8	9
0:word_views	.									x
1:visio_views		.								x
2:syntax_errors			.							x
3:semantics_errors				.						x
4:syntax_word_markers					.				x	x
5:syntax_visio_markers						.			x	x
6:semantics_word_markers							.		x	x
7:semantics_visio_markers								.	x	x
8:MarkSequence					x	x	x	x	.	x
9:ErrorToMark	x	x	x	x	x	x	x	x	x	.

(D) Error Handling Option 4

Figure 7: Design Structures using Different Error Handling Options

informal analysis results. The evolution vector can be extended with additional dimensions, such as net option values, having the potential to bridge the gap between software design modeling and rigorous economic analysis.

7. REFERENCES

[1] R. Arnold and S. Bohner. *Software Change Impact Analysis*. Wiley-IEEE Computer Society Pr, first edition, 1996.

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

[3] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. The genvoca model of software-system generators. *IEEE Software*, 11(5):89–94, Sept. 1994.

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

[5] Y. Cai and S. Huynh. An evolution model for software modularity assessment. In *Proceedings of the First Workshop on Assessment of Contemporary Modularization Techniques (ACoM)*, 2007.

[6] Y. Cai and K. Sullivan. Simon: A tool for logical design space modeling and analysis. In *20th IEEE/ACM International Conference on Automated Software Engineering*, Long Beach, California, USA, Nov 2005.

[7] Y. Cai and K. Sullivan. Modularity analysis of logical design models. In *21th IEEE/ACM International Conference on Automated Software Engineering*, Tokyo, JAPAN, Sep 2006.

[8] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, 1st edition edition, Jun 2000.

[9] J. B. Dugan, K. J. Sullivan, and D. Coppit. Developing a high-quality software tool for fault tree analysis. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 222–31, Boca Raton, Florida, 1–4 Nov. 1999. IEEE.

[10] S. D. Eppinger. Model-based approaches to managing concurrent engineering. *Journal of Engineering Design*, 2(4):283–290, 1991.

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

[12] E. Freeman, E. Freeman, B. Bates, and K. Sierra. *Head First Design Patterns*. O'Reilly Media, 1st edition edition, Oct 2004.

	0	1	2	3	4	5	6
0:word_view	.		x	x	x		
1:visio_view		.	x	x	x		
2:syntax_errors	x	x	.			x	x
3:semantics_errors	x	x		.		x	x
4:MarkSequence	x	x			.	x	x
5:excel_view			x	x	x	.	
6:xml_view			x	x	x		.

(A) Error Handling Option 1 with New Views

	0	1	2	3	4	5	6
0:word_view	.		x	x			
1:visio_view		.	x	x			
2:syntax_errors	x	x	.		x	x	x
3:semantics_error	x	x		.	x	x	x
4:MarkSequence			x	x	.		
5:excel_view			x	x		.	
6:xml_view			x	x			.

(B) Error Handling Option 2 with New Views

	0	1	2	3	4	5	6	7	8	9	1	1	1	1
0:word_views	.		x	x										
1:visio_views		.	x	x										
2:syntax_errors	x	x	.		x	x	x	x	x	x	x	x	x	x
3:semantics_errors	x	x		.	x	x	x	x	x	x	x	x	x	x
4:syntax_word_markers			x	x	.				x					
5:syntax_visio_markers			x	x		.			x					
6:semantics_word_markers			x	x			.		x					
7:semantics_visio_markers			x	x				.	x					
8:MarkSequence			x	x	x	x	x	x	.				x	x
9:excel_views			x	x						.				
10:xml_views			x	x							.			
11:syntax_excel_markers			x	x								.		
12:syntax_xml_markers			x	x									.	
13:semantics_excel_markers			x	x					x					.
14:semantics_xml_markers			x	x					x					

(C) Error Handling Option 3 with New Views

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0:word_views	.									x						
1:visio_views		.								x						
2:syntax_errors			.							x						
3:semantics_errors				.						x						
4:syntax_word_markers					.				x	x						
5:syntax_visio_markers						.			x	x						
6:semantics_word_markers							.		x	x						
7:semantics_visio_markers								.	x	x						
8:MarkSequence					x	x	x	x	.	x			x	x	x	x
9:ErrorToMark	x	x	x	x	x	x	x	x	x	.	x	x	x	x	x	x
10:excel_views										x	.					
11:xml_views										x		.				
12:syntax_excel_markers									x	x			.			
13:semantics_excel_marker									x	x				.		
14:syntax_xml_markers									x	x					.	
15:semantics_xml_markers									x	x						.

(D) Error Handling Option 4 with New Views

Figure 8: Add New Views based on Different Error Handling Options

[13] J. A. Goguen. Reusing and interconneting software components. *IEEE Computer*, 19(2):16–28, Feb. 1986.

[14] W. G. Griswold, K. Sullivan, Y. Song, N. Tewari, M. Shonle, Y. Cai, and H. Rajan. Modular software design with crosscutting interfaces. *IEEE Software, Special Issue on Aspect-Oriented Programming, January/February 2006 (in press).*, Feb 2006.

[15] J. Hannemann and G. Kiczales. Design pattern implementation in java and aspect. 2002.

[16] 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 (ECOOP)*, Finland, June 1997. Springer-Verlag.

[17] T. G. Lane. Studying software architecture through design spaces and rules. Technical Report CMU/SEI-90-TR-18, CMU, 1990.

[18] C. V. Lopes and S. K. Bajracharya. An analysis of modularity in aspect oriented design. In *AOSD '05*, pages 15–26, New York, NY, USA, 2005. ACM Press.

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

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

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

[22] D. V. Steward. The design structure system: A method for managing the design of complex systems. *IEEE Transactions on Engineering Management*, 28(3):71–84, 1981.

[23] K. Sullivan, Y. Cai, B. Hallen, and W. G. Griswold. The structure and value of modularity in software design. *SIGSOFT Software Engineering Notes*, 26(5):99–108, Sept. 2001.

[24] K. Sullivan, W. Griswold, Y. Song, and Y. C. et al. Information hiding interfaces for aspect-oriented design. In *ESEC/FSE '05*, Sept 2005.

[25] K. J. Sullivan, J. B. Dugan, and D. Coppit. The Galileo fault tree analysis tool. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, pages 232–5, Madison, Wisconsin, 15–18 June 1999. IEEE.