

# An Evolutionary Approach to Software Modularity Conformance Checking

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

## Abstract

People have realized the importance of modularity for decades, and still grapple with modularity decay caused by unexpected dependencies. Given a design that is well-modularized, it is difficult to determine whether its implementation realizes the designed modularity without introducing unexpected dependencies between modules. Manually comparing source code modular structure with abstracted design modular structure is tedious and error-prone. In this paper, we present an automated approach to check the conformance of source code modularity to the designed modularity. Our approach uses design structure matrices (DSMs) as a uniform representation. We use existing tools to automatically derive DSMs from the source code and design, and use a genetic algorithm to automatically cluster DSMs and check the conformance. We applied our approach both to a small, but canonical, software systems and to a real project. Our experiments show that automatically checking the conformance between source code and design structure has the potential to manifest the decoupling effects of design rules in source code, and to detect unexpected dependencies caused by incorrect implementation. We also show that the design model and implementation model together provide a comprehensive view of system modular structure.

## 1. Introduction

Modularity in design, determined by the dependency structure, is immensely important for software systems [2, 20]. Modularity decay caused by unanticipated dependencies hinders software evolution and maintenance [10]. Given a design that is well-modularized at, we need to detect whether the source code implements the modular structure the designer has in mind, without introducing unexpected dependencies between modules that can potentially cause modularity decay.

According to Parnas's information hiding principle [17] and Baldwin and Clark's design rule theory [3], the key step to decompose a system into modules is to determine the design rules (interfaces) that decouple otherwise coupled design decisions and to hide the decisions that are likely to change into independent modules. However, design rules and modules have different semantics at design level and implementation level.

Some design level design rules can be explicitly mapped to program level *interfaces*, such as abstract data types and function signatures. In many cases, however, design rules can be implicit, such as the data structure agreed among modules [17]. Design rules can also be crosscutting, such as naming conventions, or remain to be informal agreements among the designers. Design level *modules* are also different from *modules* at program level, which are usually equal to classes, packages, and directory structures. A module at design level, for example, can be a feature that is a vertical slicing through source code directory structure and class boundary.

Given the different perspectives of design and implementation in terms of design rules and modules, it can be difficult to determine if the source code faithfully implements the designed modularity. Unexpected dependencies can be introduced between modules during the implementation, the existence of design rules is usually blurred, and the decoupling effects of design rules to create independent modules is hard to be manifested. These difficulties make it hard to both understand the relation between design and implementation, and to detect and prevent modularity decay.

In this paper, we present an approach to automatically check the conformance of source code modular structure to design modular structure. To perform the conformance analysis, we first need a uniform representation capable of capturing the modular structure of both source code and high level design. We observe that *design structure matrix* (DSM) [21, 6, 3] is effective in fulfilling this need because it has been used to model both design and source code dependency structures [3, 22, 19].

Given design and implementation DSMs, we first formalize the conformance checking problem mathematically, and then use a genetic algorithm [8] to automate the clustering of variables in a DSM and assess the conformance between design modularity and source code modularity. A genetic algorithm is a search technique in the field of evolutionary computing, which finds approximate solutions to optimization problems by simulating the evolution of a population of potential solutions. Given two DSMs, one at the design level and the other at the source code level, our genetic algorithm takes one DSM as the optimal goal and searches for a best clustering method in the other DSM that maximizes the level of homomorphism between the two DSMs.

We developed a tool, called Glusta (Genetic Algorithm based clustering), to implement the algorithm. Using Glusta, we evaluated our approach against both a small, but canonical example, and a real project. Our experiment shows that our approach has the potential to manifest the decoupling effects of design rules on source code, to identify unexpected dependencies caused by incorrect implementation, and to provide an integrated and comprehensive view of design and implementation modularity.

The rest of this paper is arranged as follows. Section 2 introduces design structure matrices at design level and source code level. Section 3 lays out the mathematical representations and foundations to the conformance checking problem. Section 4 elaborates on the technique of the genetic algorithm. In section 5, we present our experiment on the small but canonical example. Section 6 presents our experiment on a real project. Section 7 discusses related work, and section 8 concludes.

## 2. Design Structure Matrix

Using matrices to represent dependencies has been explored by many researchers [21, 6, 3, 22, 4]. DSM models for design and source code have different but related semantics. In the next subsections, we introduce *design DSM*, the *model DSM* derived from a design DSM, the *source code DSM*, and the *conformance DSM* derived from the source code DSM.

### 2.1. Design Level DSM

The design structure matrix (DSM) was initially created by Steward [21], and later developed by Eppinger et al. [6] to model interactions between design variables of engineered systems. A DSM is a square matrix, in which the rows and columns are labeled with design variables. A cell is checked if and only if the decision on the row depends on the decision on the column.

According to Baldwin and Clark [3], design rules, the decisions that decouple otherwise coupled design decisions, can be modeled as variables aggregated into the first block of a DSM. Modules are captured by the blocks along the diagonal. If two modules are decoupled by design rules, there should be no dependencies between the two blocks of the DSM, and they both depend on design rules. DSM modeling allows the designer to cluster variables in different ways, each representing one view of decomposition. For example, Sullivan et al. [22] showed that by clustering a DSM into an environment cluster, a design rule cluster, and hidden modules, Parnas's information hiding criterion can be precisely captured.

To address the problem that manually-constructed design DSMs can be ambiguous and error-prone, Cai [4, 5] has developed constraint-based design model, called an *augmented constraint network* (ACN), that enables automatic DSM derivation and other analysis. In an ACN, design decisions and their dependencies are modeled by a logical *constraint network*, and the concept of design rules and multiple clustering are formalized as additional data structures. The ACN framework provides precise definition of *pair-wise dependence relation* (PWDR): if  $x$  depends on  $y$ , then if  $y$  changes and makes the constraint network inconsistent,  $x$  is involved in one of the ways to compensate for the change of  $y$ . Given the derived PWDR and a clustering, a DSM can be automatically derived using the supporting tool, Simon [4]. In our experiments, we use Simon DSMs to model design level modular structure.

### 2.2. Source Code DSM

Using matrix model to represent source code dependencies are also widely explored. For example, Lattix [1], a commercial tool developed by the Lattix company [1], allows designers to automatically recover a DSM from an existing software implementation. MacCormack [14] has used matrix models to compare the modular structure evolution of Mozilla and Linux. These tools extract dependencies relations among classes, including class inheritance, references, and method invocations.

There are tools that can generally model dependencies both within and among classes, such as Bunch [15] and Dependency Finder [24]. These tools can extract dependencies among variables and methods, among methods within a class, etc. Although these tools do not generate a matrix model directly, building a model based on the extracted dependencies is straightforward. In our experiments, we developed a small tool to convert the dependencies extracted by Dependency Finder into a DSM.

## 2.3. Model DSM and Conformance DSM

Given a uniform matrix representation, we can check the conformance between design and implementation modularity. It is noticeable that usually there is no one-to-one variable mapping between source code DSM and design DSM, especially when the design DSM also models environmental conditions and abstract design rules that cannot be mapped to program constructs. Accordingly, we perform the checking between source code DSM and a subset of the corresponding design DSM.

Since we are interested in the modular structure of the design and implementation, we first independently cluster both design and source code DSMs into modules. In the design DSM, we can use Simon to manually cluster variables using different ways, each reflecting a different perspective of the designer's view of modularity. For example, all the decisions about a function can be a module, each feature can be a module, and all the design rules aggregated together can also be a module. Given a clustering, we derive a *model DSM* in which each variable represents a cluster.

For the source code DSMs, which could contain hundreds of variables, manually clustering is tedious and error-prone. In this paper, we mathematically formalize the conformation problem, and introduce an AI-based evolutionary computing approach to automatically cluster the source code DSM into modules, and find the best clustering that matches the model DSM. From the optimal solution, a *conformance DSM* can thus be derived, in which each variable presents a clustered set of source code variables.

We then check the modularity conformance by examining the differences between the model DSM and conformance DSM. In the design DSM derived by Simon, the dependency relation models all possible ways a variable can depend on the other. By contrast, the source code DSM reflects one of the many possible ways that is actually implemented. Accordingly, if the conformance DSM is a subset of the model DSM, we conclude that the implementation conforms to the design modularity.

Checking the conformance helps to understand and identify the following issues: (1) if there exists a dependence in the conformance DSM, but not in the corresponding model DSM, it implies a dependence between modules that is not anticipated. (2) If in the design DSM, all the decisions about a feature are aggregated together, finding the corresponding conformance DSM helps to understand how a feature is implemented, which could crosscut directory structure of the source code. (3) If a design rule is implemented as program constructs, the conformance checking has the potential to distinguish them from other program constructs and manifest their decoupling effects.

## 3. Problem Representation

This section presents the mathematical formulation of the conformance checking problem that will be processed by the genetic algorithm.

First, we define  $\mathbb{X}$  be the set of all components in the implemented software system; that is, the set of all variables from the source code DSM.  $D_X \subseteq \mathbb{X} \times \mathbb{X}$  is the set of dependencies among the software components, such that  $\forall i, j \in \mathbb{X} (i, j) \in D_X$  if and only if component  $i$  depends on component  $j$ . That is  $(i, j) \in D_X$  if and only if the cell representing the dependency of variable  $i$  on variable  $j$  is marked in the source code DSM. Analogously, we define  $\mathbb{Y}$  and  $D_Y$  in regards to the set of all variables in the model DSM.

DSMs can be considered as attributed, directed graphs in which the variables serve as attributes for the vertices and the dependency structure portrayed by the DSM serve as directed edges. That is, each vertex of the graph represents a variable of the DSM and there exists a directed edge in the graph if and only if there is a marked cell in the DSM between those two variables. Accordingly, we transform both the source code DSM and the model DSM into graphs for purpose of mathematical processing.

In the next subsections, we first defined one of the two DSMs as a *sample graph*, then define the other as a *model graph*, and finally define their *conformance criteria*.

### 3.1. Sample and Model Graphs

A *sample graph* has more vertices than a model graph and needs to be clustered; it is usually transformed from a source code DSM. Mathematically, a sample graph is defined as an attributed, directed graph  $\vec{S} = (V_S, E_S)$  (i.e.  $V_S$  is the set of vertices and  $E_S$  is the set of edges). Attributes are assigned to each vertex to associate it with a particular variable of the source code DSM via the bijection  $k_S : V_S \rightarrow \mathbb{X}$ . There is an edge in the graph between two vertices if and only if the corresponding variables have a dependency (i.e.  $E_S \equiv D_X$ ). A *model graph*, is defined the same was as the sample graph but usually in relation to the model DSM. More formally, a model graph is defined as  $\vec{M} = (V_M, E_M)$  in relation to  $\mathbb{Y}$  and  $D_Y$ .

### 3.2. Conformance Criteria

To determine the conformance of the source code modularity to the high level design modularity, we find a graph homomorphism from the sample graph to the model graph. First, we cluster the variables of the sample graph and therefore form a new graph  $\vec{C} = (V_C, E_C)$  (i.e.  $V_C$  is the set of vertices and  $E_C$  is the set of edges), called the *conformance graph*. Each vertex of the conformance graph is associated

with a cluster of variables from the sample graph. This is rigorously defined via the bijection  $k_C : V_C \rightarrow V_S / \sim$ , such that the clustering of variables forms a quotient set of  $V_S$  with respect to an arbitrary equivalence relation  $\sim$ . Edges are defined by the dependencies among components within each equivalence class such that  $\forall v_i, v_j \in V_S / \sim (v_i, v_j) \in E_C \Leftrightarrow \exists (u_i \in v_i, u_j \in v_j) (u_i, u_j) \in E_X$ .

The more conforming the source code modularity is to the design modularity, the closer to isomorphic the conformance graph and model graph will be. In assessing the level of isomorphism between two graphs, the *graph edit distance* [18] is computed between the graphs. The graph edit distance ( $\delta$ ) is defined as the number of edit operations needed to transform one graph into the other. For this paper, the edit operations are limited to node insertion/deletion and edge insertion/deletion. Given two graphs,  $G_1$  and  $G_2$ ,  $\delta(G_1, G_2) = 0$  only when the graphs are isomorphic.

## 4. Evolutionary Computing

With the given representation of the problem, we formulate a genetic algorithm [8] whose goal is to find the projection  $\pi^* : V_S \rightarrow V_C$  to produce  $\vec{C}$  such that  $\delta(\vec{C}, \vec{M}) \approx 0$ . Meaning that we want to find a homomorphism from the sample graph to the model graph; in doing so the conformance graph is isomorphic to the model graph. If a homomorphism does not exist, then we want to find a mapping that is as close to homomorphic as possible. It is known that the graph homomorphism problem is NP-Complete [7], hence there are no known efficient solutions. Therefore, we use a non-deterministic approach by employing a genetic algorithm.

In a genetic algorithm, potential solutions to the problem are considered as organisms with genetic material, similar to DNA. In this paper, we refer to these potential solutions as projections. Each projection  $\pi_i$  is a sequence  $\langle \alpha_1, \alpha_2, \dots, \alpha_{|X|} \rangle$  of mappings for each vertex of the sample graph to a vertex of the conformance graph. Each  $\alpha_j$  is an element from  $V_C$  and the projection maps the  $j$ -th vertex in  $V_S$  to it. The algorithm first creates an initial population  $\pi_1, \pi_2, \dots, \pi_n$  of random projections.

Each projection is judged on how accurate its solution to the problem is, called its fitness. Pairs of projections are selected for “breeding” to produce new projections. These new projections are created by combining the genetic material from their “parent” projections. Only the projections with the highest fitness values survive long enough to breed and pass on their genetic information. The algorithm breeds these projections, one iteration at a time, until a satisfactory solution is found or until a maximum iteration count is reached.

## 4.1. Fitness

Designers can cluster a DSM in different ways, each representing a perspective of decomposition: directory structure, horizontal layers or vertical slicing [5]. To check the conformance between design and implementation when the design DSM is clustered in a particular way, we need to cluster the source code components accordingly. One feature of a genetic algorithm based approach is that we can configure its *fitness function* to reflect the designer’s intentions to cluster a system so that variables within a sample DSM can be clustered accordingly.

In this section we discuss how the fitness function can be configured to control the way the source code components are clustered. Each projection is assigned a fitness based on how accurate its solution is to the problem. For our current approach, the fitness of each organism is calculated using the formula:

$$-\delta(\vec{C}_i, \vec{M}) - \epsilon(\vec{C}_i, \vec{M}) - \lambda(\vec{C}_i, \vec{M}) - \phi(\vec{C}_i, \vec{M}) \quad (1)$$

where  $\vec{C}_i$  is the conformance graph at the  $i$ -th iteration of the algorithm.

The fitness function component  $\delta$  is the graph edit distance, modeling the differences between the dependency structures of the two DSMs. If the graph edit distance is zero, the conformance graph and model graph are isomorphic. In most cases, we can’t expect the source code structure to be identical to design structure. Accordingly, our approach does not require the graph edit distance to be zero for the algorithm to terminate. Instead, our algorithm finds the solutions that maximizes the level of isomorphism.

In this paper, we assume that all the design dimensions modeled in a model DSM are implemented in the source code. Based on this assumption, we augment the fitness function with a penalty for projections in which the conformance graph vertices and model graph vertices do not have a one-to-one mapping. We define such a penalty using  $\epsilon$ , as shown in formula (1).

Using the graph edit distance for the fitness function is not sufficient. Our initial experiments showed that there may be multiple ways to cluster the sample DSM that create the same dependency structure. In our algorithm, we add two additional components to the fitness function to provide finer differentiation between projections with the same graph edit distance. These two functions allow us to configure a sample graph so that it can be clustered in different ways, each corresponding to how the design targeted DSM is clustered.

### 4.1.1. Directory Groupings

It is often the case that in large software systems, source code components belonging to the same high level design

component are grouped into the same file system directory or package. For example, Lattix derives DSMs from Java source code and organizes the derived DSMs so that each block along the diagonal represent a Java package. Accordingly, our approach allows the user to specify the directory groupings of source code components. We use a dissimilarity metric to calculate how separated components from each directory grouping are. Depending on this measure, the fitness of the projection is proportionally reduced. This reduction of the fitness is denoted in formula (1) as  $\lambda$ .

#### 4.1.2. Name Patterns

Using regular expressions to model name patterns to capture a group of components is a well understood approach. Murphy's Relexion [16] uses name patterns to specify mappings between high level models and implementations; in *aspect-oriented programming* (AOP) paradigm, name patterns are used to specify *join points* for the weaving of advice into aspect code. In many cases, name patterns exist in source code representing commonalities between different components [9].

Accordingly, our approach allows the user to specify name patterns as part of the fitness function such that sample graph vertices matching the pattern should be clustered and mapped to a particular model graph vertex. If a sample graph node attribute matches the pattern but is not correctly mapped to the model graph vertex then the fitness of the projection is reduced. This reduction of the fitness is denoted in formula (1) as  $\phi$ .

## 5. A Canonical System

We apply our approach to automatically checking the modularity conformance between the design and implementation of Parnas's canonical *Keyword in Context* (KWIC) system [17]: "*The KWIC index system accepts an ordered set of lines; each line is an ordered set of words, and each word is an ordered set of characters. Each line is circularly shifted by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order*". In his seminal paper [17], Parnas presents two KWIC designs: a traditional sequential design (SQ) based on the sequence of converting the input to the output, and a new one based on information hiding (IH).

Our recent work [4, 5] has formalized both the SQ and IH designs as ACNs and generated the design DSMs using Simon. Being as famous as it is, KWIC has been implemented by many people. We selected, by random search, a version developed at the Institute for Information Processing and Computer Supported New Media (IICM), Graz University

of Technology<sup>1</sup>. Given the source code, we use an open source tool, *Dependency Finder* [24], to extract detailed dependency information for both the SQ and IH implementations.

### 5.1. Making Design Rules Explicit

In the SQ design, Parnas describes five modules: Input, Circular Shift, Alphabetizing, Output, and Master Control. We model each module using three decision variables: signature, data structure, and implementation. Parnas also analyzes the significant impact of the following environmental conditions: the input format, input size, core size and alphabetizing policy. We similarly model these environmental conditions as variables, and model the relations among these variables using logical constraints. The variables and constraints constitute the constraint network.

According to Parnas: "*The defining documents would include a number of pictures showing core format, pointer conventions, calling conventions, etc. All of the interfaces between the four modules must be specified before work could begin.*" [17]. Accordingly, we model procedure signature and data structures as *design rules*, represented by a *dominance relation* augmenting the constraint network. We also modeled the different ways to cluster a design into modules as a *cluster set*. The constraint network, dominance relation, and cluster set constitute the SQ augmented constraint network (ACN). Please refer to [5] for the full SQ ACN built in Simon.

Using Simon, we generated the SQ DSM clustered in different ways, each representing a different perspective of modularity. Figure 1 shows a Simon DSM in which all the variables related to a function is aggregated together; Figure 2 shows a DSM with the same set of dependencies but are clustered in a different way: all the data structures, part of the design rules, are clustered as a module.

From Figure 1, we can see that the SQ design are highly coupled. According to Parnas, if data structures can be determined beforehand, the implementation of each function still can be done independently. This is captured in Figure 2: after all the data structures are aggregated into a design rule module, all the other functions become independent modules, shown as blocks along diagonals. To understand how the data structure design rules are implemented into the source code, we need to derive a model DSM from the design DSM shown in Figure 2, generate sample DSM, and using Glusta to find the best mapping.

Because the source code does not contain any environmental information, the environmental variable in the design DSM do not participate the comparison. We also excluded the Master Control module because it has trivial effects. Accordingly, we only consider the subset of the de-

<sup>1</sup>[http://coronet.iicm.edu/sa/swp\\_how.htm](http://coronet.iicm.edu/sa/swp_how.htm)

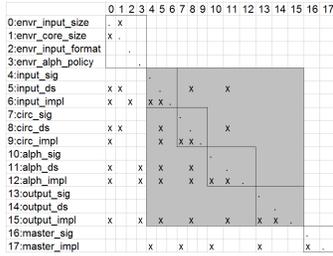


Figure 1. Design DSM

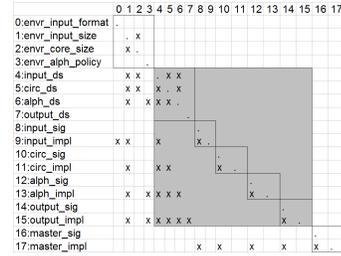


Figure 2. Design DSM with DRs

sign DSMs (the gray block of Figure 2) for the purpose of conformance checking. Figure 3 (A) shows the derived model DSM.

We now generate the sample DSM from the source code. We extract the dependency information using Dependency Finder, and convert the result into a sample DSM. The source code includes one Java class with five functions: Master Control (main), Input, Circular Shift, Alphabetizing, and Output. In the sample DSM, the variables, including both data members and methods of the KWIC class, are randomly ordered, and the dependencies among these variables and methods are marked.

Given the model DSM (Figure 3 (A)), Glusta automatically finds the best solution that aggregated the sample DSM into the order as shown in Figure 3 (B). Glusta also generates the corresponding conformance DSM, which is exactly the same as the model DSM, meaning that the implementation conforms to the designed modularity.

## 5.2 Integrated Modularity View

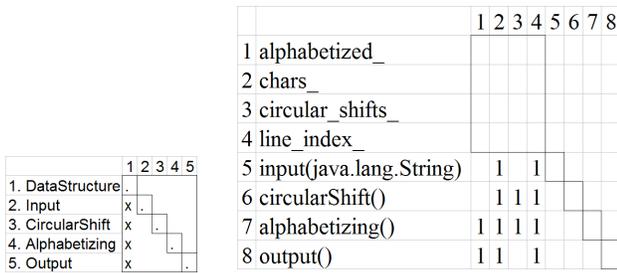
In KWIC information hiding (IH) design, Parnas splits the line storage function from the input and gives each its own interface, hence the information hiding (IH) design has one more model DSM variable, *LineStorage*. The environment variables are modeled the same way as in the SQ ACN. The logical relations are changed accordingly, and the data structure variables are no longer design rules.

Figure 4 shows the design DSM derived and clustered using Simon. To facilitate comparison, we aggregate all the variables related to a function into a module, and we expect the dependencies among these modules are no more than the dependencies shown in its model DSM (Figure 6(A)).

Given the model graph, Glusta automatically found the best solution that clusters the sample DSM as shown in Figure 5, and computed its conformance DSM as shown in Figure 6(B). By comparison, we can conclude that the implementation conforms to the designed modularity in that the conformance DSM is a subset of the model DSM.

Having a closer look at the two dependencies that exist in the model graph, marked in gray cells, but not in the conformance graph, we found that both Alphabetizer and Output depend on LineStorage, which is correct according to the semantics of the design. These dependence are not shown in the conformance DSM. After inspecting the source code, we found that the CircularShifter class uses LineStorage as a member, Alphabetizer uses a CircularShifter as a member, and in turn, Output uses a Alphabetizer as member. As a result, the Alphabetizer class calls the interfaces of LineStorage *indirectly*. The lack of syntactic reference between CircularShifter and Alphabetizer results in the lack of dependencies in the code-based model. However, the CircularShifter's functions called by Alphabetizer, and the Alphabetizer's functions called by the Output, are all defined in LineStorage. If the definition of these functions changes in LineStorage, all the other functions have to be changed.

We observe that viewing the model DSM and conformance DSM together reveals a more comprehensive dependency structure: both direct and indirect dependencies can



(A) Model DSM (B) Clustered Sample DSM

Figure 3. KWIC SQ Model and Sample DSMs

From 3 (B), we can clearly identify the program constructs corresponding to the design rules. In this case, the four data members, *chars\_*, *line\_index\_*, *circular\_shifts\_*, *alphabetized\_*, are the design rules crosscutting and governing the other functions. This helps us understand how the SQ design is implemented without closely examining the source code.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0:envr_input_format	.																			
1:envr_input_size		.	X																	
2:envr_core_size			X	.																
3:envr_alph_policy				.																
4:linestorage_ADT					.															
5:linestorage_ds			X	X		.	X													
6:linestorage_impl			X	X	X	.														
7:input_ADT							.													
8:input_impl			X		X		X	.												
9:circ_ADT									.											
10:circ_ds			X	X						.	X									
11:circ_impl					X		X	X	.											
12:alph_ADT										.										
13:alph_ds			X								.	X								
14:alph_impl			X	X			X	X			X	X	.							
15:output_ADT														.						
16:output_ds															.	X				
17:output_impl			X					X			X	X	.							
18:master_ADT																			.	
19:master_impl			X		X	X			X		X			X					X	

Figure 4. Design DSM

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1 LineStorage																				
2 Input																				
3 CircularShift																				
4 Alphabetizing																				
5 Output																				

Figure 5. Sample DSM

	1	2	3	4	5
1. Linestorage	.				
2. Input	X	.			
3. CircularShift	X		.		
4. Alphabetizing	X		X	.	
5. Output	X		X	.	

(A) Model DSM

	1	2	3	4	5
1. Linestorage	.				
2. Input	X	.			
3. CircularShift	X		.		
4. Alphabetizing			X	.	
5. Output				X	.

(B) Conformance DSM

Figure 6. KWIC IH Conformance Checking

be modeled and distinguished. In addition, if we plug the conformance DSM into the design DSM, we can observe directly which environment conditions influence which class, linking the source code structure with design structure and environment conditions directly.

### 5.3 Performance

We ran our tool on the both KWIC SQ and IH to check the consistency between design and implementation. Our experiments consistently converged to produce the desired result. We configured our genetic algorithm to run 30 generations, and correct results are constantly found within 20 seconds on our machine. Although currently our program only allows the user to specify directory groupings and name patterns, the fitness function can be extended with additional criteria.

## 6. Case Study: Liebeherr’s HyperCast

HyperCast [11, 12] is a scalable, self-organizing overlay system developed using Java, a project with hundreds

of files. Its key abstraction is the socket supporting point-to-point and multicast communication in overlay networks. HyperCast integrates overlay sockets, viewed as nodes, into networks in a decentralized manner. It also offers network services including naming, reliable transport and network management. Key dimensions in HyperCast design include the following:

- *Socket*: the overlay socket APIs
- *Protocol*: the protocols for maintaining various network topologies
- *Monitor*: a network management capability
- *Service*: provides services such as end-to-end service, naming service, etc
- *Adapter*: a layer that virtualizes underlying networks
- *Logging*: a mechanism to record selected events; logging of informational messages, of raised exceptions, and of errors that do not raise exceptions
- *Events*: implicit invocation that several HyperCast modules use to notify clients of key events

Sullivan et al. [23] presented a comparative analysis of three HyperCast designs: the actual OO design, an “oblivious” AO design produced by moving scattered code into aspects, and a design rule (DR) design based on information hiding interfaces. To test the applicability of our approach to real project, we ran Glusta on the DSMs of the OO and DR design and implementation. For the sake of space, we only report our analysis of the DR design and implementation.

For each design, we similarly use Simon [4] to generate the design DSM. Given the source code, both Lattix and Dependence Finder can extract the dependencies among classes. We export and convert the extracted dependencies into the input of Glusta.

To specify the directory groupings for the fitness function, we created a tool to automatically report which classes belong to each Java package. Several naming patterns were specified to improve the fitness function of the genetic algorithm. We renamed some variables in the source code to reflect the designer’s intentions regarding which module a class belongs to. This is because the although standard naming conventions were used, not all all parts of the code followed these conventions.

In the design rule (DR) design, Sullivan et al. [23] established a set of crosscutting design rules, serving as information hiding interfaces to decouple the base code design and the aspect-oriented design. These design rules constrain the execution phenomena that must be exposed as join points, how they are exposed (e.g. naming convention, syntax, stack shapes, etc.) and the behavior across join points (e.g. pre- and post-conditions for the execution of advice compositions).

In Simon, we similarly abstracted them as design variables (from 11 to 17 in Figure 7 ), and modeled the fact that both the basic and crosscutting variables respect these design rules as logical constraints. The DSM derived by Simon contains 33 variables, modeling specifications, abstract design rules, and implementations (Figure 7). The DSM shows that there are no dependencies between the basic function variables and crosscutting variables. Instead, they both depend on the abstract design rules.

From the 33 variables, we take a subset and cluster it in to 7 clusters. We include the design rule module because most of these design rules can be mapped to HyperCast APIs, which follows the naming convention that the Java Interface starts with “I\_”. The generated model DSM contains 7 variables (Figure 9 (A)).

The designers have moved fragments from the original code to aspects, and refactored the original code in ways dictated by the design rule interfaces. In the sample DSM generated for the refactored code, there are 206 variables. The genetic algorithm was run several times for 200 generations, taking approximately 40 minutes each run. Consistently, the algorithm would produce the same homomorphism and when manually checked, this homomorphism was consistent with the designer’s intentions. The sample DSM clustered by Glusta is shown in Figure 8. The generated conformance DSM is shown in Figure 9(B).

The best edit distance calculated by Glusta is -12, meaning that no perfect isomorphism is found. By examining the model DSM (Figure 9(A)) and the conformance DSM (Figure 9(B)), we observe a number of discrepancies.

The “x” marks in the dark cells of the model DSM indicate that the Monitor module depends the Protocol, Service, and Adapter, but these dependencies are not shown in the conformance graph. This is because the Monitor module exchange data with these other modules through external XML files, which can not be captured by a static analysis tool.

The “?” marks in the conformance graph show several dependencies that do not exist in the model graph. The designer of HyperCast confirmed that these source code dependencies are caused by incorrect or imperfect implementation. All of the discrepancies in row 1 are caused by implementers improperly using other components that are not interfaces. For example, the *IInterceptionCallback* interface uses the *OLMessage* class in its definition when it should have used the *IMessage* interface instead. The remaining of differences in column 3, were caused during the refactoring procedure, when the author wrote some experimental code that should have been removed.

The “?” marks in column 7 of the conformance graph show that the functional modules depend on the aspect modules. This is caused by AspectJ dynamic weaving. After the aspect code is weaved into Java classes, a static analyzer will reveal that the classes depends on the aspects. From the design level, these dependencies are already removed due to introduction of the seven design rules.

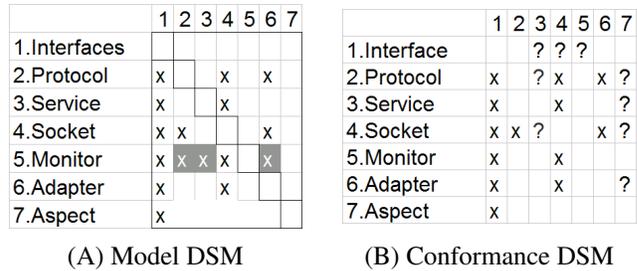


Figure 9. HyperCast Conformance Checking

In summary, by checking the modularity conformance automatically, we can identify the dependencies that are not detected by static analysis tools, identify both indirect dependencies and direct dependencies, identify dependencies that are removed at design level but still exist in code, and discover unexpected dependencies caused by incorrect or imperfect implementation that could later on cause maintenance problems.

Our experiment also shows that source code analysis alone does not appear to be sufficient for one to understand the designers intention and how the system is modularized. In particular, when external files and dynamic weaving are involved, the extraction becomes more complicated. On the other hand, by checking the conformance between design modularity and implementation modularity, we can obtain a

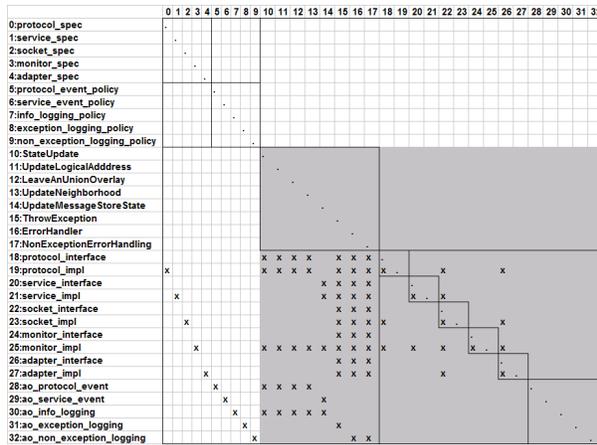


Figure 7. Design DSM



Figure 8. Sample DSM

more comprehensive view of the overall dependence structure.

## 7. Related Works

DSMs were created by Steward [21], developed by Eppinger [6], and has been applied to other engineering disciplines [6, 21]. Baldwin and Clark have sketched a novel theory providing new insights into the connections between design structure and economic value in design using DSM modeling [3]. Sullivan et al. [22, 23], and Lopes et al. [13] have shown that DSM modeling is valuable for software design. For example, a DSM can precisely capture Parnas’s information hiding criterion [22].

There are several other approaches and tools for automatic clustering and analyzing software modularity. Typical “bottom-up” tools, such as Bunch [15] and Lattix [4], reverse engineer high level design models from source code. Our automatic clustering approach is different in that we use a high level design model as the optimal goal and our fitness function is configurable. Design level modeling tools, such as Simon [4], support high level logical design modeling, and is not connected with implemented source code.

Our work is similar to Murphy’s Reflexion Model [16] in that our approach also combines a source code model with a high level model. Our work is different from Murphy’s work in that our approach employs the power of DSM, supports auto-clustering and multi-clustering, and emphasizes modularity checking. Our approach has the potential to link source code structure with design structure and environment conditions, so that we can extend Baldwin and Clark’s net option value analysis [3] and Parnas’s information hiding analysis [17] to the level of source code.

## 8. Conclusion

To address the problem that a well modularized design does not always guarantee a well modularized implementation, and the difficulty of manually checking the conformance between a design modular structure and a source code modular structure, we presented an approach to automatically check the consistency between source code structure and design structure. Our approach makes use of design structure matrices as uniform representations, uses existing tools to automatically derive design level and implementation level DSMs, employs a genetic algorithm to cluster the DSMs and to check the their consistency by finding the best mapping.

We have applied our approach to both a small, but canonical, *keyword in context* software system and a real project, HyperCast. Our tool, Glusta, automatically clusters the DSMs and check the consistency between design and implementation DSMs. The conformance checking has the potential to identify design rules that are implemented as program constructs, to identify unexpected dependencies that may indicate false implementation, and to enable a maintainer to understand the design and implementation comprehensively.

## 9. Acknowledgments

We thank Kevin Sullivan for inspiring the idea of comparing design DSM with source code DSM. We especially thank Yuanyuan Song, one of the designers and implementers of HyperCast, for helping us understand and analyze HyperCast.

## References

- [1] *The Lattix Approach Whitepaper*. Lattix Inc., Nov. 2004.
- [2] C. W. Alexander. *Notes on the Synthesis of Form*. Harvard University Press, 1970.
- [3] C. Y. Baldwin and K. B. Clark. *Design Rules, Vol. 1: The Power of Modularity*. The MIT Press, 2000.
- [4] 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*, Nov. 2005.
- [5] Y. Cai and K. Sullivan. Modularity analysis of logical design models. In *21th IEEE/ACM International Conference on Automated Software Engineering*, Tokyo, JAPAN, Sept. 2006.
- [6] S. D. Eppinger. Model-based approaches to managing concurrent engineering. *Journal of Engineering Design*, 2(4):283–290, 1991.
- [7] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman, 1979.
- [8] J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, 1992.
- [9] 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.
- [10] M. M. Lehman. *Program Evolution: Processes of Software Change*, chapter 12, pages 247–274. Academic Press, London, UK, 1985.
- [11] J. Liebeherr and T. K. Beam. Hypercast: A protocol for maintaining multicast group members in a logical hypercube topology. In *Networked Group Communication*, pages 72–89, 1999.
- [12] J. Liebeherr, M. Nahas, and W. Si. Application-layer multicasting with delaunay triangulation overlays. *IEEE Journal on Selected Areas in Communications*, 20(8), Oct. 2002.
- [13] 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.
- [14] A. MacCormack, J. Rusnak, and C. Baldwin. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Harvard Business School Working Paper Number 05-016.*, 2004.
- [15] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. *IEEE Proceedings of the 1999 International Conference on Software Maintenance*, Aug. 1999.
- [16] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflection models: Bridging the gap between source and high-level models. In *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 18–28, 1995.
- [17] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–8, 1972.
- [18] A. Sanfeliu and K.-S. Fu. A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, 13(3):353–362, 1983.
- [19] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *OOPSLA*, 2005.
- [20] W. P. Stevens, G. J. Myers, and L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–39, 1974.
- [21] 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.
- [22] 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.
- [23] K. Sullivan, W. Griswold, Y. Song, and Y. C. et al. Information hiding interfaces for aspect-oriented design. In *ESEC/FSE '05*, Sept. 2005.
- [24] J. Tessier. *The Dependency Finder User Manual*.