A Case Study of Integrating Design Rule Theory and Analytical Decision Models Into a Software Development Process

Yuanfang Cai and Sunny Huynh
Dept. of Computer Science
Drexel University
Philadelphia, PA, 19104 USA
{yfcai, sunny}@cs.drexel.edu

Abstract

The interdependencies of design decisions, made at various stages of software development, determine the modular structure in design, and hence the maintainability and evolvability of the implemented software product. It has been understood that the key step to decompose a system into modules is to identify the design rules that decouple otherwise coupled design decisions. Emerging analytical decision models, such as design structure matrices (DSMs) and augmented constraint networks (ACNs) have been used to explicitly model design rules and to analyze software modular structure. In this paper, we present a case study of integrating the design rule theory and these analytical decision models into the development process of a real project, to govern the creation and realization of design rules and to monitor the modular structure of the system throughout the development life cycle. Our experience shows the feasibility and benefits of the integrated process: this integration allowed us to detect poorly-modularized design before coding, to make the design rules and their decoupling effects explicit in both design and implementation, and to detect improper implementation by checking modularity deviation between design and source code.

1. Introduction

Software designers make decisions at various stages of software development. The interdependencies of these decisions crosscut multiple software artifacts, influence the modular structure in design, and affect the maintainability and evolvability of the implemented software. It has also been understood that the key step to decompose a system into modules is to determine the interfaces–*design rules* in Baldwin and Clark's modularity theory [4]–so that the modules become independent from each other and only depend on the interfaces [17, 4, 22, 23]. The quality of design rules thus determines the quality of the resulting modular structure. Prevailing software modeling techniques, such as the Unified Modeling Language (UML), are not designed to govern the creation, realization, and evolution of design rules, nor to monitor and maintain software modular structure throughout the development process.

Two emerging analytical decision models that explicitly and formally model design rules and design modular structures are *Design Structure Matrices* (DSMs) [21, 4, 23, 22, 12, 14, 19] and *Augmented Constraint Networks* (ACNs) [7, 8, 6, 11]. Large-scale DSM models have been used to conduct source code modularity analysis and evolution analysis [14, 12, 19]. DSM modeling is at the heart of Baldwin and Clark's influential design rule theory, showing how design rules create independent modules and option values [4]. Sullivan et al. introduced DSM modeling to software design, showing that a DSM can precisely capture Parnas's information hiding criterion [23, 22].

The *Augmented Constraint Network* (ACN) [7, 8, 6, 11] is the logic-based counterpart of the DSM at design level, developed to address the problem that manually constructed design-level DSMs are ambiguous and error-prone. An ACN model uses a constraint network as the computational core, from which we have precisely defined the concept of *pair-wise dependency*, so that a DSM model with precise semantics can be automatically derived. We have shown that ACN modeling, supported by our prototype tool, Simon [7, 5], formalizes the key notions in Baldwin and Clark's modularity theory and Parnas's information hiding criterion, and enables us to conduct change impact analysis [7, 8] and to quantitatively compare aspect-oriented vs. object-oriented designs [6, 8]. Combining design-level ACN/DSM modeling with source code DSM has enabled us to conduct modularity conformance checking [11].

We observe that ACN and DSM modeling have the potential to be integrated with the software development pro-

cess, guiding software modularization activities, and monitoring and ensuring system modular structure throughout the development life cycle. In this paper, we report our experience of integrating DSM and ACN models into the development process of a real online financial management system, which employs a service-oriented architecture (SOA) and allows the user to manage financial transactions, financial accounts, user accounts, etc.

In this case study, we derive ACN and DSM models from the software artifacts generated during the standard development process, such as UML models and requirement specifications. Based on the ACN/DSM models, we are able to (1) check the quality of design in terms of modularity before coding; (2) monitor the creation of design rules at design level and their realization in the source code; (3) check the conformance between design and implementation. These activities can be continuously applied when the software evolves and during the maintenance stage, for example, to evaluate and check the effectiveness of refactoring activities [12, 14].

Our experiment obtained positive and encouraging results. Identifying the *design rules* and making their decoupling effects explicit contributed to the final delivery of a well-modularized product. Using the integrated process, we were able to detect poorly-modularized design and incomplete test plan before coding, and to detect incorrect implementations before testing. Our case study provides preliminary guidelines about how to integrate design rule theory and the supporting analytical decision models into a software development process, and justifies the further automation of this integrating process.

The rest of this paper is organized as follows. Section 2 introduces the concept of design rules, design structure matrices and augmented constraint networks. Section 3 overviews our integrated process. Section 4 presents our results of applying the framework to a real software development process. Section 5 discusses lessons learned and future work. Section 6 discusses related work and Section 7 concludes.

2. DR Theory and Decision Models

The key concepts used in this paper are the *design rule theory* and two supporting analytical decision models, the *Design Structure Matrix* (DSM) and *Augmented Constraint Network* (ACN).

Design Rules. According to Baldwin and Clark's modularity theory [4], *design rules* are stable architectural design decisions that decouple otherwise coupled design decisions. Design rules in software can be data formats agreed among development teams [17, 23], naming conventions [22], or interfaces between components [12]. The essence of the design rule concept is the *dominance relations* among design decisions and their *splitting* effects: design rule decisions *dominate* other decisions in that they should not be affected by these subordinate decisions; design rules *split* subordinate decisions into *modules* in that these modules will only depend on the design rules, and not on each other. As a result, each module becomes independently substitutable as long as the design rules remain stable. The key step to create a modular structure is thus to create the design rules, and to split the system into modules accordingly. Both DSM and ACN modeling can explicitly capture the dominating and decoupling effects of design rules, and the resulting modular structure.

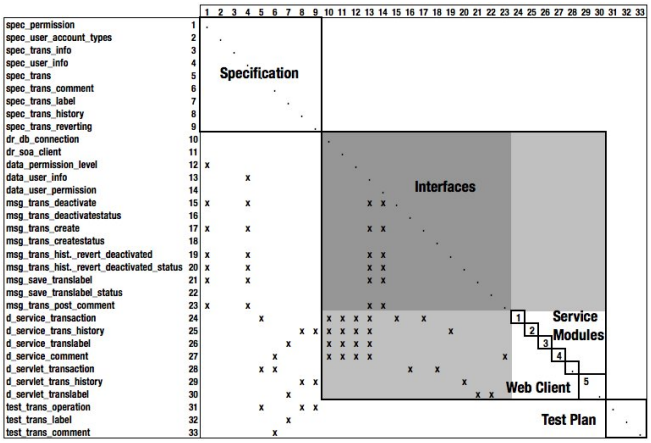


Figure 1. Design Structure Matrix

Design Structure Matrix. A *Design Structure Matrix* (DSM) presents in a square matrix form the pair-wise dependence structure of design decisions. In a DSM, the rows and columns are labeled with design variables, each representing a design dimension where a decision needs to be made. A marked cell indicates that the decision of the dimension on the row depends on the decision of the dimension on the column. Figure 1 depicts a partial DSM for the design of the online financial management system we studied, homogeneously modeling decisions made at various stages, such as requirement analysis, design and testing. For example, the marked cell in row 27, column 6 models that the decision about how to implement the function of allowing the user to add comments to a financial transaction (`d_service_comment`) depends on the corresponding specification (`spec_trans_comment`). DSMs are also used to represent source code structures [12, 14, 19]. Figure 2 depicts a partial DSM derived from the source code that implements the design as shown in Figure 1.

A DSM captures the *dominance relation* between design rules and subordinating decisions by aggregating de-

sign rule variables into a block preceding the subordinating modules, and making their dependencies asymmetric. For example, in Figure 1, the *Interfaces* block contains all the design rules of the system that should not be affected by other design variables. As a result, there are no dependency marks in the upper right gray block. Figure 1 similarly shows that the requirement specification, usually agreed by outside stakeholders, dominates all other decisions in the system. The *splitting* effects of design rules are captured by independent modules along the diagonal. For example, after clustering all the design rules into the *Interfaces* module, all the subordinate modules, labeled 1 to 5 in Figure 1, become independent in that there are no off-block dependencies among these blocks. A DSM can represent multiple clustering methods by reordering the columns and rows.

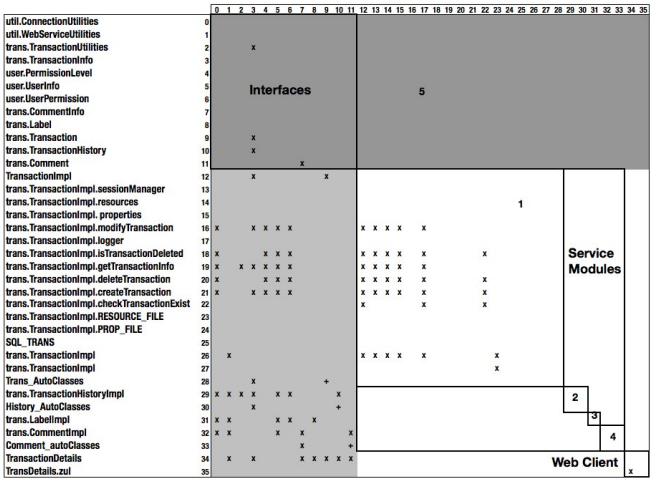


Figure 2. Source Code DSM

Augmented Constraint Network. As the counterpart of DSM at design level, our prior work presented the *Augmented Constraint Network* (ACN) [7, 5] as a formal design representation better subject to the automated analysis of design evolvability and changeability properties. Our prior work shows that the ACN modeling is more expressive and precise than the DSM modeling, and that it not only enables automatic DSM derivation, but also enables a number of additional analysis techniques, such as design change impact analysis and task decomposition [7, 5, 8].

The core of an ACN is a finite-domain *constraint network* (CN) [15], which consists of a set of *design variables* modeling design dimensions. Each *design variable* has a domain that comprises a set of *values*, each representing a decision or condition. For example, a SOA application can choose either dynamic proxy or static stub as the client-service communication mechanism, which can be modeled as the following scalar design variable: dr_soa_client : $\{dynamic, static\}$. By contrast, a DSM only shows design dimensions, but not the concreted choices with each dimension. A design decision is represented by a binding of a *value* from a *domain* to a *variable*. For example, $dr\_soa\_client = dynamic$ models the choice of dynamic proxy. We model dependencies among decisions as logical expressions. The following expression shows that the design of the transaction web client (`d_servlet_transaction`) in the second version (`v2`), *assumes* that dynamic proxy is employed. Logically, the binding of the assuming variable implies the assumed binding [7, 5].

```
d_servlet_transaction = v2 => dr_soa_client = dynamic;
```

To capture the dominating effects of design rules and the concept of *modules*, we augment a constraint network with a binary *dominance relation* and a *cluster set* data structure. For example, the decision on `dr_soa_client` is the one of the design rules of the online financial management system, and the pair, (`d_service_transaction, dr_soa_client`), is a member of the dominance relation of the project ACN, indicating that the transaction service design should not influence the communication mechanism determined by the architect. We use a *cluster set* to model the fact that the same design can be viewed in different ways. Each *cluster* in the set consists of a set of variables organized into a tree structure, modeling one way the system can modularized. The constraint network, dominance relation, and cluster set constitute an *Augmented Constraint Network*.

We have shown that ACN modeling formalizes all the key concepts captured by DSM modeling, and a DSM can be automatically generated from an ACN [7]. The basic idea is to formally define the semantics of *pair-wise dependency* based on ACN: if x depends on y, then there must be a change in y that makes the constraint network inconsistent, and at least one of the minimally disruptive way to compensate for the change involves a change in x. We have created an algorithm to automatically compute all the dependency pairs in an ACN to form a *pair-wise dependence relation* (PWDR). A DSM can thus be automatically derived by using these pairs to populate the matrix and using a selected cluster to order the columns and rows.

Our prior work [8] presented an approach that uses the non-trivial *dominance relations*, that is, the formalized design rules, to *split* a large ACN model into a set of smaller sub-ACNs. This divide-and-conquer approach addresses the scalability issue caused by the difficulty of constraint solving. We observe that each sub-ACN, created by design rules, has the potential to be an independent responsibility assignment module. We can derive a sub-DSM from each sub-ACN, representing the structure of a sub-system. Our prior work also presented the algorithm of integrating the sub-DSMs into a full DSM [8]. The user can also choose to analyze each sub-system individually or to combine a selected set of sub-DSMs into a partial DSM. As a result, we

can in principle apply all the analysis capabilities developed for DSMs to our much more complete and precise models.

In summary, DSM and ACN modeling differ from prevailing software modeling techniques in that they both can generally model design decisions made at various stages of software development process. They capture the essence of design rules, their decoupling effects, and the resulting the modular structure.

Analysis Enabled by Decision Models. Our prior work shows that ACN and DSM modeling can enable a number of analysis techniques, as supported by Simon [7, 5]:

(1) *Design changeability analysis.* If one of the decisions changes, for example, the SOA communication mechanism is changed to use static stubs in the online financial management system, the designers/implementers need to know the impact of this change. Using Simon, the user can perform change impact analysis introduced in our previous paper [7, 5]. Given a changing decision as input, Simon will compute which and how many decisions need to be revisited. We have shown that this approach [7] has fully formalized and quantified Parnas's changeability analysis in his seminal paper [17].

(2) *Information hiding modularity.* Sullivan et al. [23] present a characterization of the nature of Parnas's information hiding modularity as invariance of design rules with respect to changes in environment variables. Our ACN model has fully formalized these concepts [7, 5]. In essence, given a design-level DSM derived from an ACN, or a source code DSM derived from the implementation, after aggregating the environmental parameters and design rules into separate modules, we expect to see the following *modularity pattern* if the system follows Parnas's information hiding criteria [17]: there are no dependencies from design rules to environmental parameters; there are no dependencies from design rules to other subordinating variables; and the subordinate modules are independent from each other, showing as independent blocks along the diagonal [12, 23].

(3) *Module uniformity.* After a large project ACN is decomposed into a number of smaller sub-ACNs, we expect all the design dimensions within a sub-ACN are closely related to a particular function, following the separation of concerns and similarity of purpose principles [1, 20].

(4) *Modularized conformance checking.* Given that we can uniformly model both design and source code as DSMs, we have developed an approach to automatically check the conformance between designed and implemented modular structures [11]. Moreover, after we decompose the design into sub-ACNs, we are able to check the consistency of each pair of sub-models at a much smaller scale, that is, comparing a design-level sub-DSM derived from a sub-ACN, and a source-level sub-DSM derived from the source code.

3. Integrated Process Overview

Our purposes of integrating the ACN/DSM models into a standard software development process are to maintain a well-modularized structure in both design and implementation, and to govern the creation, realization, and evolution of design rules. Figure 3 shows a standard software development process integrated with the analyses enabled by these models. The left column shows standard software development activities, from requirement analysis to software testing. Each of these activities generates corresponding software artifacts, such as requirement specification, UML diagrams, and source code. We use Lattix [13] to reverse engineer a source DSM from the source code, and use an ACN model to represent decisions in other software artifacts. From an ACN, we automatically derive a design DSM. The integration of analytical decision models are independent of the development process model in use, which can be either waterfall, iterative, or agile.

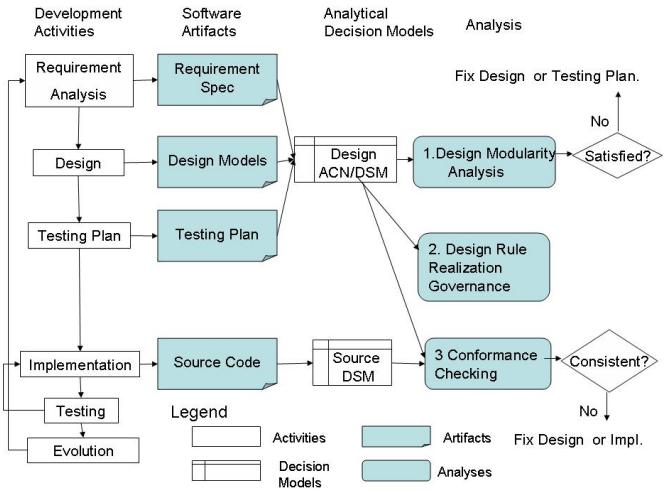


Figure 3. Integrated Framework Overview

Design Modularity Analysis. After the requirements are agreed, and the design is modeled using prevailing modeling techniques, we want to make sure that the design is well-modularized before investing in coding. Our first step is to model all the decisions in the artifacts into an ACN, and explicitly identify and record design rules in the dominance relation of the ACN. After that, we use Simon to decompose the full-ACN into sub-ACNs based on the formalized design rules. We then evaluate the sub-ACNs to see if they follow the separation of concern and similarity of purpose principles [1, 20]. We also check the completeness of the design and test plan to see if all the required functions are designed and included in the test plan.

Design Rule Realization Governance. The importance of making important implicit design decisions explicit has been recognized. Given the design rules created in the previous step, we record how these design rules are mapped into the source code. The source code elements corresponding to design-level design rules become the design rules in the implementation. Making their dominating roles explicit helps avoid accidental changes to these elements, enables the user to check if the implementation also follows the *modularity pattern* as introduced in Section 2, and eases the conformance checking between design and implementation.

Conformance Checking. After the design is implemented, we want to make sure that the implementation does not introduce any unexpected dependencies and faithfully implemented the designed modular structure. We first reverse engineer a source DSM from the source code, check the modularity pattern in it, and automatically compare it with the design DSM derived from the design ACN [11]. If there is an inconsistency, we examine the reason and fix the design or implementation if necessary.

These analyses can be continuously applied when the software evolves and during the maintenance stages. For example, if a refactoring plan or a new feature is proposed, we can use the *Design Modularity Analysis* technique to make sure the proposed new design will not cause modularity decay. After a refactoring plan or a new feature is implemented, we can check to see whether *ex post* outcomes match the *ex ante* goals of the effort [12] using the *Conformance Checking* technique.

4. A Case Study

This section presents our experience of integrating the ACN/DSM models into a real online financial management project. The project we study is a financial management system developed for the Student Activity Fee Allocation Committee (SAFAC) at Drexel University. SAFAC is in charge of the fund allocation to over 160 student organizations, and needs to randomly perform audits on these organizations each year. A senior design group in the Computer Science department proposed to develop a web-based application, *VODKA Organizational Device for Keeping Assets* (VODKA), to standardize the financial management method among all student organizations, facilitate financial activity tracking, provide a central storage of all organizations' financial information, and simplify the auditing.

Following the standardized senior design project guidelines, the development process went through the following stages: requirement analysis, design, implementation, and testing. At each stage, the students were asked to deliver a corresponding artifact, including: a requirement specification agreed by the stakeholder, a design specification containing UML models, a test plan, and final source code. The team followed a hybrid iterative and agile methodology: whenever a problem in design or implementation was found, the team would quickly redesign or reimplement the system. The project involved six team members and took about six months.

From the artifacts generated from the process, the authors derived the decision models and conducted the analysis independently. The rest of the team was not aware of the design rule theory nor the modeling techniques. Once the analysis techniques revealed a problem, such as a design defect or improper implementation, the authors would consult the team members to see if the analysis result was valid. All the artifacts generated during the process were also peer-reviewed by other senior design teams. The purpose of this case study was to see if the integrated process could detect design and implementation problems that were not detected by the peer-review process.

4.1. Design Modularity Analysis

As introduced in the previous section, the design modularity analysis consists of the following steps: modeling decisions into an ACN model, creating and formalizing design rules, and analyzing design modular structure using Simon.

Modeling Decisions into an ACN. We first derive a constraint network from the requirement specification, UML design models, and test plan. The VODKA requirements specification document is written strictly conforming to IEEE standards [9], and approved by SAFAC. The document specifies both functional and non-functional requirement items, and each item is prefixed with an index number for later reference. For example:

```
1060 The system shall allow users to attach any file to
     a specified transaction. Any number of files maybe
     attached to a transaction limited to the hardware
     resources.
```

We view each item in the specification as a *variable*. Take item 1060 for example, this item specifies the functionality of attaching files to transaction records. The concrete requirement could change over time, for example, from "any number of files" to a particular number of files for some unforeseen reason. Accordingly, we generally view "file attachment" as a dimension where the designers have to make a concrete decision, and model it using a variable `spec_trans_attachment`. The prefix `spec` indicates that this is a specification decision. We model the concrete decision of this dimension as the *values* of this variable, and all the values constitute the *domain* of this variable. Since the item 1060 in this version of specification has specified the decision, we model this decision as value `v1_1060`. Because there can be other choices for file attachment, we

generally model these unknown choices as `other`. As a result, we formally model the file attachment requirement as: `spec_trans_attachment: {v1_1060, other}`.

There are 168 indexed items in the VODKA requirement specification, including the creation, deletion, modification of user accounts, financial accounts, financial transactions, as well as other functions such as report generation and email notification. We abstractly model them using 40 specification variables.

The design document describes and models the combined three-tier and service-oriented architecture design using architectural diagrams, UML sequence diagrams, entity-relation diagrams, and database schemas. We derive ACN models from the UML sequence diagrams, which depict in detail the services that will be provided by the system and their relations. Figure 4 shows a partial sequence diagram of the financial transaction creation function, from which we derive design decisions as follows:

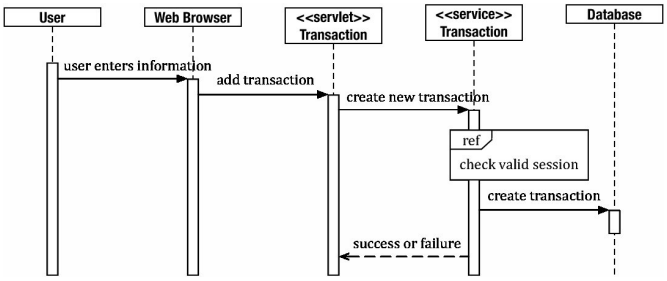


Figure 4. Sequence Diagram of the Transaction Creation Function

- Each service (labeled with `<<service>>`) and its web client (labeled with `<<servlet>>`) in the sequence diagrams are design dimensions that the designers decide to include. The detailed design of each service or web client can vary. Accordingly, we view each of them as a *variable*. For example, the transaction creation function described in section 2.7.3 of the first version design document is modeled as: `d_service_trans_create: {v1_2_7_3, other}`.

- The messages passed between the services and their clients are also design dimensions. The decisions for these dimensions are particularly important because they are the *interfaces* between difference components. For example, the "*create new transaction*" message that goes from `<<servlet>>Transaction` to `<<service>>Transaction` can be viewed as the contract between the service `Transaction` and its web client, containing the necessary transaction information in a format agreed between them. Both the concrete formats and content of these messages are subject to change. Accordingly, we abstract messages as variables, and generally model the possible choices of each using two values `orig` (short for original) and `other` (some future unelaborated choices). For example, `msg_trans_create:{orig, other}` models the transaction creation message.

There are 24 sequence diagrams in the first version design document of VODKA, and we use 121 variables to model the decisions in them. We similarly derive testing decisions from the acceptance test plan document. Each section of the plan describes the preconditions, postconditions, and expected actions for each function. The following variable models the testing decision about the transaction creation function specified in section 6.2.4 of the acceptance test plan document, version 1: `test_trans_create: {v1_6_2_4, other}`.

After modeling these decisions using variables, we then model their dependence relations as logical expressions. Some of the dependencies are derived from the same artifact, such as the dependency between the messages and services shown in a sequence diagram:

```
d_service_trans_create = v1_2_7_3 => msg_trans_create = orig
```

Other dependencies may crosscut multiple artifacts, such as the dependencies between design decisions and requirement decisions:

```
d_service_trans_create = v1_2_7_3 =>
    spec_trans_create = v1_0910
```

In summary, we model the first version of VODKA using a constraint network with 162 variables. The model is extensible in the sense that new design components can be added and decisions such as database schema can also be modeled, if needed.

Identifying Design Rules. The key part of the design process is to determine the *design rules* (DRs) so that the system can be modularized and the big ACN can be decomposed into smaller sub-ACNs. We distinguish three types of DRs in VODKA, architectural design rules, data structure design rules, and interface design rules.

Architectural design rules. These are the design decisions that influence the whole project. Following are two examples:

1. `dr_data_storage:{database,file,other}` models the choice of how to store the financial data of the system. The VODKA designers decided to use a database, modeled by `dr_data_storage = database`.

2. `dr_soa_client:{dynamic,static,other}` models the type of web service clients to use: whether the clients are bound to the services at compile time or at run time. The VODKA designers decided to use dynamic proxy, that is, `dr_soa_client = dynamic`.

Data structure design rules. The requirement specification dictates some highly crosscutting information, such as the permission level and user information for

the financial system. These information are designed as data structures and modeled using the following variables: `data_permission_level:{v2,other}` models the *permission level* data structure in the second version of the design, and `data_user_info:{v2,other}` models the *user info* data structure.

Interface design rules. These design rules specify the interfaces between web clients and services, that is, the messages passing between them. The variable `msg_post_comment : {orig,other}` is an example. Its decision determines the format and content of the message.

We then formalize these design rules into the dominance relation of the VODKA ACN. The dominance relation also dictates that requirement decisions dominate other decisions. Finally, we cluster requirement decisions, design rule decisions, and decisions for each services and web-clients into modules. The constraint network, dominance relation, and the cluster constitute a big ACN model.

Design Modularity Analysis. Based on these design rules, we first decompose the big ACN into a number of smaller sub-ACNs, generate sub-DSMs, and compose them into a full VODKA DSM. As an example, Figure 5 shows the decomposed sub-ACN modeling the function of adding comments to transactions. From this sub-ACN, we can easily identify all the decisions needed for the function and the constraints among them. For example, the sub-ACN shows that the function is corresponding to the requirement specified in item 1030; the implementer needs to know the user's information and permission level; the comment format is as agreed (`msg_post_comment=orig`), etc.

```
designspace d_service_comment{
    spec_user_info:{v1_0280,other};
    spec_permission:{v1_0590,other};
    spec_trans_comment:{v1_1030,other};
    dr_data_storage:{database,file,other};
    dr_soa_client:{dynamic,static,other};
    data_permission_level:{v2,other};
    data_user_info:{v2,other};
    msg_post_comment:{orig,other};
    d_service_comment:{v2,other};

    d_service_comment = v2 =>
        spec_financial_trans_comment = v1_1030
        &&  msg_post_comment = orig
        && dr_soa_client = dynamic
        && dr_data_storage = database
        && data_permission_level = v2
        && data_user_info = v2;
    data_user_info = v2 =>
        spec_user_info = v1_0280;
    data_permission_level = v2 =>
        spec_permission = v1_0590;
}
```

Figure 5. Transaction Comment ACN

We inspect each sub-ACN to see if the decisions it contains serve similar purposes, if it contains some requirement dimensions that the functions represented in the sub-ACN intend to fulfill, and if it contains some testing dimensions indicating that these functions will be tested according to the test plan. We expect the sizes of the ACNs to be evenly distributed as much as possible. Table 1 lists the functionalities and sizes of all the sub-ACNs derived from the first version of VODKA.

We identified two main issues. First, the sizes of two sub-ACNs are extremely large: the transaction service (4) and its GUI (18). Looking into these sub-ACNs, we found that too many requirements are handled by these modules. In addition to handle transaction creation, deletion, modification, these modules also have to handle transaction history recording and reverting, user comments, and labeling. Second, several sub-ACNs do not contain testing decisions, which means that the test plan is not complete.

Table 1. The Decomposition of the First Version of VODKA

no.	module	size	no.	module	size
1	attachment	3	12	fin acct (ui)	3
2	fin acct.	14	13	login (ui)	3
3	notification	3	14	notification (ui)	3
4	transaction	31	15	report (ui)	12
5	user acct.	17	16	search (ui)	4
6	login	12	17	summary (ui)	11
7	logout	3	18	transaction (ui)	30
8	report	13	19	history (ui)	4
9	revert	3	20	user acct. (ui)	10
10	session	11	21	view (ui)	20
11	trend	4			

We reported our results to the developers of the team, and we were confirmed that these were design defects that they were not aware of and were not pointed out by the peer-review process. The team then redesigned the system, for example, by adding additional services to handle transaction history, comments, etc., and making the test plan complete. We similarly modeled the second version of the design into an ACN and decompose it into 27 sub-ACNs. Table 2 shows their sizes and functions.

From the table, we can see that the second version has more modules with smaller sizes, and the sizes of each are more evenly distributed. We checked each sub-ACN to make sure its variables serve similar purposes. Since the analysis was done *before coding*, the team was able to redesign easily and avoid the more expensive cost of recoding.

4.2. Design Rule Realization Governance

Table 3 shows several sample design rules and their realizations in the source code. For example, the DR

Table 3. The Design Rule Realization

Sample Design Rules	Source Code Realization	Comment
`dr_data_storage = database`	`ConnectionUtilities` Class	JDBC database connections
`dr_soa_client = dynamic`	`WebserviceUtilties` Class	Setting up dynamic proxies
`data_permission_level = v2` `data_user_info = v2` Data Structure	`PermissionLevel` Data Structure `UserInfo` `UserPermission` Data Structure	Data structures visible to all components
`msg_post_comment = orig`	`CommentInfo` Data Structure `Comment` Interface	Interface between the web-client and service

Table 2. The Decomposition of the Second Version of VODKA

no.	module	size	no.	module	size
1	acct. label	3	15	acct. label (ui)	3
2	authentication	6	16	fin. acct (ui)	9
3	comment	5	17	login (ui)	7
4	fin. acct.	7	18	notification (ui)	3
5	login	10	19	report (ui)	10
6	notification	4	20	search (ui)	3
7	report	10	21	summary (ui)	10
8	session	6	22	history (ui)	6
9	history	7	23	transaction (ui)	8
10	transaction	16	24	trans label (ui)	5
11	trans. label	4	25	user acct. (ui)	4
12	trend	5	26	user label (ui)	3
13	user acct.	13	27	view (ui)	18
14	user label	4			

`msg_post_comment = orig` is translated into a `Comment` interface and a `CommentInfo` data structure, which are the design rules at the source-code level, play a key role in the implementation.

First, both the services and the clients will have to know the interface in the model-view-controller pattern. For example, the `edu.drexel.cs.vodka.trans.CommentImpl` in Row 32 of Figure 2 is the comment service implementation, which implements that `Comment` interface. `transdetails.zul` in Figure 2 is the view and the `TransactionDetails` is the controller. The `TransactionDetails` controller invokes the service through the `Comment` interface, and has to know the `d_soa_client` design rule to call the web service using dynamic proxy.

Second, we run the `wsimport` tool on these interfaces, which generates two supporting classes for each method signature and several other supporting classes. From example, one method in the interface `Comment` is `getComments`, for which the tool generates two supporting classes `getComments` and `getCommentsResponse`. These classes are used by the GlassFish web server to support the dynamic proxy invocation technique. Therefore, `CommentInfo` data structure and `Comment` interface, realizing the `msg_post_comments` design rule, dominate how these supporting classes are generated.

As a result, these design rules dominate other components and split the client and service implementation completely. Using Simon, we can derive a sub-DSM from each sub-ACN, combine then into a full DSM, or combine a selected set of sub-ACNs to form a partial DSM. Figure 1 shows a partial DSM derived from all the sub-ACNs related to transactions: transaction creation, transaction history, transaction comments, and transaction labeling.

4.3. Conformance Checking

After the system is implemented, we use Lattix [13] to reverse engineer the source code into a source DSM. Since Lattix puts dependents on the column while Simon puts dependents on the row, we use a small tool to transpose the Lattix DSMs, making them comparable to design DSMs derived from Simon. Figure 2 shows the transaction part of the VODKA source DSM derived and converted from Lattix.

From the source DSM, we first check if the implementation follows the modularity pattern, making sure that there are no unexpected dependencies between modules or from the design rules to subordinate modules. After that, we check the conformance between design and implementation by comparing design DSMs derived from the ACN with the source DSM. Although the implementation follows the modularity pattern, we found incorrect implementations through the conformance checking.

We conducted the conformance checking on each pair of sub-models, that is, comparing each design-level sub-DSMs with a corresponding source-level sub-DSM. Figure 6 (A) depicts a DSM derived from the sub-ACN shown in Figure 5. Figure 6 (B) shows the corresponding source-level sub-DSM derived using Lattix [13]. These DSMs model the transaction commenting function of VODKA.

By comparing Figure 6 (A) and (B), we found that dependency between the design of the comment service (`d_service_comment`) and permission level (`data_permission_level`) is not realized in the source DSM: the cell with dark background and "?" indicates the missing dependency.

We then inspect the `CommentImpl` service, and found that the implementer of this service forgot to check if the user has the permission to add comments to the transaction, which is a small function but with significant security consequences.

The '+' sign in Figure 6 (B) indicates an implicit dependency that is not detected by Lattix, the dependency from `Comment_autoClasses`, the classes automatically generated by `wsimport`, to the `Comment` interface. This dependency indeed exists because `wsimport` is manually run and generates these classes according to the interface, although the generated classes do not refer to the interface explicitly. We thus manually added this kind of dependencies, using '+' signs.

5. Lessons Learned

In summary, integrating the design rule theory and analytical decision models in the project enabled us to find poorly-modularized design and incomplete test plan before coding, and to detect implementation defects such as missing functionality before testing. Although all the design documents were also peer-reviewed by other senior design teams, none of these problems were identified by them.

The key lesson is to make the design rules explicit at early stages of the development process. Making DRs explicit at design-level provides all the team members a clear picture of the architectural decisions that they need to respect throughout the life cycle. Making the DRs explicit at the source code level makes it clear to the implementers that these design rule classes should not be changed or depend on their own implementation.

Second, decomposing the system into sub-ACNs benefits the development process. Each sub-ACN models an independent task assignment so that each implementer only needs to know the design rules that affect his tasks before he can start the implementation independently. These sub-ACNs also enable us to check the quality of the design modularity before coding, to inspect the design of each module individually at a much smaller scale, and to finally conduct a modularized conformance checking without the necessity of dealing with huge models.

Third, integrating automatic modularity analysis techniques and conformance checking appears to be necessary and useful. Detecting design defects by examining lengthy documents is difficult. Discovering improper implementations, such as missing a critical but small functionality, usually requires complete test cases and a thorough testing procedure. Our integrated process has the potential to discover these defects early in the process to avoid the more expensive cost of recoding.

Our experience also reveals several issues that lead to our future work. First, as a feasibility study, the VODKA ACNs were generated manually, which is time consuming and still subject to inaccuracy. We observe the opportunity of automatically generate ACNs from exiting artifacts. One of our current work [10] formally defines the dependency semantics in UML class diagrams and automatically derives ACNs from these models.

Second, the VODKA case study does not include the modeling and analysis of test cases, which is another future work. Finally, VODKA is not an industrial-level large-scale project. The scalability and usability of applying logical models into large-scale software systems need to be further evaluated. On the other hand, although VODKA is not large, it followed standard development procedure, and underwent relatively strict review, testing, and grading process. The problems identified and addressed by the design rule theory and the analytical decision models, such as poorly-modularized design, can only be more prominent in larger projects.

6. Related Work

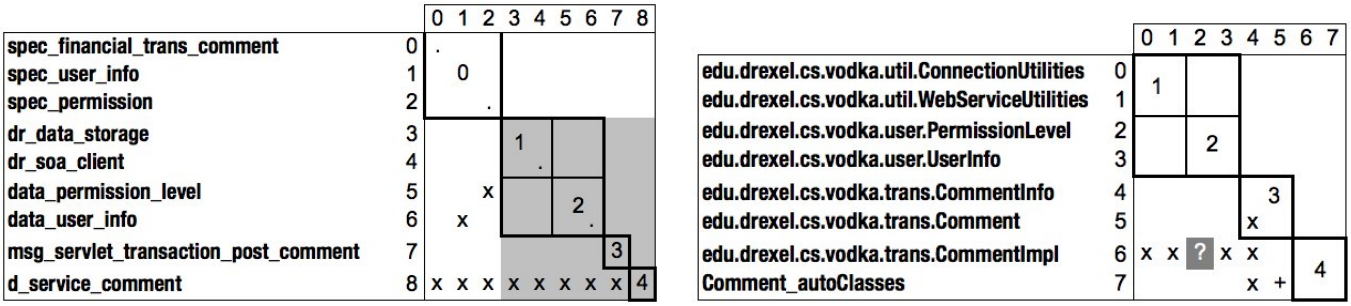
Our work is related to process-oriented traceability analysis [3, 16, 18]. Asuncion et al. [3] present an end-to-end traceability tool that also handles overall software development life cycle, in which heterogeneous artifacts are stored into database. Neumuller et al. [16] also report the success of using databases to store and manage the links between software artifacts. Our work is different in that we extract decisions from heterogeneous software artifacts and uniformly represent them as analytical decision models, enabling system decomposition and modularity analysis.

Anderson et al. [2] similarly employ an approach that translates heterogeneous artifacts to a homogeneous representation, and perform traceability analysis based on the same representation. By contrast, our uniform decision representation formalizes the key notions of design rule theory, and supports automatic decomposition.

Similar to the work of Pohl [18] and Asuncion et al. [3], our framework also integrates analysis with software development practice. Different from their work, we emphasize the development of design rules as the key part of the development process.

7. Conclusion

In this paper, we presented our experience of integrating design rule theory and the two supporting analytical models, ACN and DSM, into the development process of an online financial management system. The integrated process enabled us to monitor the creation and realization of design rules, and to maintain a well-modularized structure in both design and implementation. Using these models, we detected poorly-modularized design and incomplete test plan before coding, and detected modularity deviation caused by



(A) Transaction Comment Design

(B) Transaction Comment Implementation

Figure 6. Modular Conformance Checking

missing functionality in the implementation before testing. Our experiment shows potential positive impacts of the integrated process, justifying the further development of automated approach of deriving analytical decision models from existing software artifacts.

References

[1] E. B. Allen, T. M. Khoshgoftaar, and Y. Chen. Measuring coupling and cohesion of software modules: An information-theory approach. *metrics*, 00:124, 2001.

[2] K. M. Anderson, S. A. Sherba, and W. V. Lepthien. Towards large-scale information integration. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 524–534, New York, NY, USA, 2002. ACM Press.

[3] H. U. Asuncion, F. François, and R. N. Taylor. An end-to-end industrial software traceability tool. In *ESEC-FSE '07*, pages 115–124, New York, NY, USA, 2007. ACM Press.

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

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

[6] Y. Cai, S. Huynh, and T. Xie. A framework and tool supports for testing modularity of software design. In *22nd IEEE/ACM International Conference on Automated Software Engineering*, Nov. 2007.

[7] 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.

[8] Y. Cai and K. Sullivan. Modularity analysis of logical design models. In *21th IEEE/ACM International Conference on Automated Software Engineering*, pages 91–102, Tokyo, JAPAN, Sep 2006.

[9] S. E. S. Committee. Recommended practice for software requirements specifications. (830), 1998.

[10] S. Huynh, Y. Cai, and W. Shen. Automatic transformation of UML models into analytical decision models. Technical Report DU-CS-08-01, Drexel University, 2008.

[11] S. Huynh, Y. Cai, Y. Song, and K. Sullivan. Automatic modularity conformance checking. In *Proceedings of the 30th International Conference on Software Engineering*, May 2008.

[12] M. J. LaMantia, Y. Cai, A. D. MacCormack, and J. Rusnak. Analyzing the evolution of large-scale software systems using design structure matrices and design rule theory: Two exploratory cases. In *7th Working IEEE/IFIP International Conference on Software Architectures*, pages 83–92, Feb. 2008.

[13] Lattix Inc. *The Lattix Approach Whitepaper*. Lattix Inc., Nov. 2004.

[14] A. MacCormack, J. Rusnak, and C. Y. Baldwin. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science*, 52(7), July 2006.

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

[16] C. Neumuller and P. Grunbacher. Automating software traceability in very small companies: A case study and lessons learne. *21st IEEE International Conference on Automated Software Engineering*, 0:145–156, 2006.

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

[18] K. Pohl. *Process-Centered Requirements Engineering (Advanced Software Development Series)*. Research Studies Press, 1996.

[19] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *OOPLSA*, 2005.

[20] S. Sarkar, G. M. Rama, and A. C. Kak. Api-based and information-theoretic metrics for measuring the quality of software modularization. *IEEE Trans. Softw. Eng.*, 33(1):14–32, 2007.

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

[23] K. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. *ESCE/FSE 9*, 26(5):99–108, Sept. 2001.