

# Logic-Based Software Project Decomposition

Yuanfang Cai and Sunny Huynh

Drexel University, Philadelphia, PA, 19104, USA

**Abstract.** Prevailing design modeling techniques are not designed to generally model decisions made throughout software lifecycle, nor to support automatic modularity analysis. We have developed an analyzable design representation called an *augmented constraint network* (ACN), and an algorithm to decompose a big ACN model into smaller sub-ACNs. To evaluate the effectiveness of the decomposition algorithm in terms of enabling parallel implementation and the feasibility of modeling and analyzing software decisions made in practice using our framework, we apply our techniques to an online student society financial management project. Among other results, the experiment shows that (1) our framework is general enough to uniformly model decisions made within different project stages, such as requirement analysis and architecture design; (2) each sub-ACN, automatically decomposed using our prototype tool, Simon, corresponds to an independent responsibility assignment that can be implemented in parallel; and (3) our approach supports automatic traceability and change impact analysis.

## 1 Introduction

Prevailing software design modeling techniques are not designed to generally model software decisions and their dependence relations that span multiple stages in software lifecycle, such as requirement, design, and testing. Consequently, when the requirement changes, we lack an automatic way to find the decisions made in other stages that need to be revisited.

According to Parnas [16], a *module* is defined as a responsibility assignment that can be accomplished independently. From the perspective of overall software lifecycle, a *module* spans multiple stages. For example, to implement a service or a feature independently, the implementor needs to know the relevant subset of requirements, the relevant subset of design decisions, and the interfaces to other relevant modules. Current design modeling techniques and tools are not designed to support such module decomposition automatically.

To address these problems, we first need a design representation that is general enough to model decisions within different project stages. Recent work, such as that of Sullivan et al. [17, 18] and Lopes et al. [14], has shown that *Design Structure Matrices* (DSM) modeling is valuable in software design modularity analysis. In a DSM, decisions can be generally modeled as variables labeling the rows and columns of a square matrix, in which marked cells model the dependences among these decisions. However, a DSM does not explicitly represent

multiple decisions within a dimension, and does not support the analysis of and comparison among these decisions. Manually-constructed DSMs are also shown to be ambiguous and error-prone [7, 8].

We have developed a logic-based analyzable design model called an *augmented constraint network* (ACN) [7, 8, 5], in which design decisions and their relations can be generally modeled as variables and logical expressions of a constraint network. Different from other constraint-based design space modeling techniques, we model non-logical but critical decision-making aspects as auxiliary data structures. In particular, we model the dominance relation among decisions and multiple design views as two additional data structures: *dominance* and *cluster set*.

Using the constraint network and dominance relation of an ACN, we derive a non-deterministic automaton to capture all possible ways in which any change to any decision in any state of a design can be compensated for by minimal sets of changes to other decisions, which we call a *design automaton* (DA). DA enables basic design impact analysis [7, 5]. Summarizing from a DA, we further derive a pair-wise dependence relation (PWDR). Combining PWDR with selected cluster from the cluster set of the ACN, a DSM can be automatically derived.

We have shown that ACN modeling, supported by our prototype tool, Simon, formalizes the key notions in Baldwin and Clark's modularity theory [3] and Parnas's information hiding criteria [16]. Because DSM can be derived from ACN, ACN modeling thus has all the capability of DSMs, provides DSM modeling with precise semantics, and brings additional benefits such as the explicit representation, analysis, and comparison of different design choices.

To address the scalability issue caused by the difficulty of constraint solving, we created an approach to make use of the non-trivial *dominance* relation in an ACN model to decompose an ACN into a number of smaller sub-ACNs, solve each sub-ACN separately, and integrate partial results, but only as needed [8]. We observe that this divide-and-conquer approach not only reduces the time required to analyze the design models we studied dramatically, but also has the potential to decompose a model into independent responsibility assignment modules.

To evaluate the effectiveness of the decomposition algorithm in terms of enabling parallel implementation, to evaluate the feasibility of modeling and analyzing decisions and their relations in multiple design stages using ACNs, and to comprehensively evaluate the framework in practice, we apply our modeling and analysis techniques to a real senior design project being developed by a group of senior students in the department of computer science, Drexel University. Upon completion, the product will be used by the Activity Fee Allocation Committee (SAFAC) at Drexel to manage, track, and audit the financial activities of all the student organizations on campus.

The team strictly follows software engineering standards, going through the stages of requirement analysis, design, implementation, and testing. Agile development process is employed, and three iterative builds, each including design, coding, integration, and testing are scheduled. According to the senior design

guidelines provided by the department, the team develops an artifact at each stage, such as requirement specification and design documentation.

Given the requirement and design documents, the team faces the problem of how to decompose the tasks into modules so that the implementation can be done in parallel. Assigning tasks according to features does not guarantee parallel implementation because there is neither a clear picture about the interfaces among these features, nor about the constraints that should be respected. The documentations with hundreds of pages do not provide a desirable basis for task decomposition either. In addition, given envisioned changes, it is hard to estimate the impacts.

To address these problems on one hand and to evaluate the ACN framework on the other, we model the decisions and their relations extracted from the requirement specification, design document, and testing plan into an augmented constraint network (ACN). After that, we use Simon to automatically decompose the big ACN into a number of smaller sub-ACNs, analyze each one individually, and compose the results when needed.

We observe that each decomposed sub-ACN contains a subset of decisions in requirement, design, and testing plan respectively that are relevant to one function, forming an independent responsibility assignment *module*. The team can thus split their tasks accordingly and implement these modules in parallel. We also show that our framework is capable of modeling design decisions and design rules in contemporary service-oriented architecture uniformly, and enables automatic traceability analysis and change impact analysis.

The rest of this paper is organized as follows. Section 2 introduces our online financial management project. Section 3 illustrates how we model design decisions within different stages using an ACN. Section 4 reports our decomposition and analysis results. Section 5 discusses related work. Section 6 concludes.

## 2 Application Overview

The outside stakeholder, Student Activity Fee Allocation Committee (SAFAC) at Drexel, is in charge of the fund allocation to over 160 student organizations, and needs to randomly perform audits on organizations each year. The difficulty comes from the fact that these organizations manage their financial activities in different ways: some use old-fashioned paperwork and some use spreadsheets. To address the problem, 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 senior design project guidelines provided by the department, the team has strictly followed software engineering standards, going through requirement analysis, design, implementation, and testing stages. At each stage, the students are asked to provide an artifact, a documentation or source code.

The requirement specification conforms to IEEE standards [9] and is approved by the outside stakeholder, SAFAC. 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. The senior design team also developed a acceptance testing plan.

At this point, the implementation tasks need to be assigned to implementors for the first build and testing with a predetermined deadline. Following an agile development process, two additional rounds of design, implementation, and testing will be performed. Given the deadline, ideally the tasks should be implemented in parallel. However, the documentation does not provide a good basis for task decomposition. First, the documents are lengthy and imprecise. Second, the modeling diagrams are large and hard to read. Third, the dependence relation crosscuts the overall requirements and architectural design.

We tried to decomposed the tasks according to the horizontal layers of the architecture, such as user interface components and data processing components. However, there is no clear picture about what are the interfaces among these components, what decisions have to made before two implementors can work independently, what part of the requirements an implementor should fulfill, and what are the assumptions and constraints the implementor should respect. Moreover, the consequences of making changes are hard to predict.

As the advisor and the main designer of this system, the authors extract all the decisions from these documents and model them as variables; extract the dependence relation among these decisions and model them using logical expressions, constituting a constraint network. We also model the *dominance* relation among these decisions as an auxiliary data structure. For example, the requirement specification, as agreed by the outside stakeholder, dominates other decisions. Finally, we model the different ways the design can be viewed using another data structure called *cluster set*. The three parts, constraint network, dominance relation, and cluster set constitute an ACN.

Given the VODKA ACN, we first use the algorithm introduced in our previous work [8] to decompose the big model into 21 sub-ACNs using Simon. We observe that each sub-ACN involves all the decisions needed for one component, including parts of the requirement, parts of the design, and parts of the testing plan. In other words, each sub-ACN represents a responsibility assignment *module*. A sub-ACN, along with its automatically generated DSM, provides a compact view showing the related requirements, design rules, constraints, and testing plans, enabling independent implementation.

In addition, through analyzing each module and integrating the results, we identified several design issues that can not revealed by the informal documents only. We can also perform change impact analysis and traceability analysis using Simon. In the next sections, we illustrate how to generally model decisions within different project stages and their relations into an ACN model, and report our decomposition and analysis result.

### 3 Project Modeling

This section presents how we recognize, extract, and model software decisions and their dependencies within multiple development stages of VODKA, how to identify and model the dominant relation among these decisions, and how to model the different ways the same system can be viewed.

#### 3.1 Model Decisions as Variables

Designers make decisions at various stages. In the requirement analysis stage, the designers make decisions such as what functions will be included in the systems and what are the interfaces to external systems; in the design stage, the designers make decisions such as what classes or services should be included for which particular requirements. We now introduce how we abstract and model these decisions from the following aspects.

**Requirement Decisions** The VODKA requirements are specified strictly conforming to IEEE standards [9], and are approved by the outside stakeholder. The document specifies both functional and non-functional requirements. In this specification, each item is prefixed with an index number for later reference. The priority number following the description indicates whether the requirement will be implemented in the first release. For example, The item 1130 is a planned future enhancement. Figure 1 depicts part of the requirement documentation:

We view each item 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 account, financial account, financial transactions, as well as other functions such as report generation and email notification. We abstractly model them using 40 specification variables.

**Architectural Decisions** The general architecture of VODKA is a combination of the three-tiered and service-oriented architecture. The traditional three-tier architecture is designed to cater the needs of a central database and server so that the user can remotely manage and audit student society financial activities.

- 1060 The system allows users to attach any file to a specified transaction. Any number of files may be attached to a transaction limited to hardware resources. **Priority 1**
- 1070 The system allows each transaction to be tagged with multiple labels. **Priority 1**
- 1080 The system allows any number of labels to be created for each account, limited to hardware resources. **Priority 1**
- 1090 The system maintains a permanent history of all actions applied to transactions from a user account. **Priority 1**
- 1100 Deleted transactions are maintained in the permanent history of the financial account. **Priority 1**
- 1110 The system allows the reverting of transactions to any prior state in its history by any user with the privilege. **Priority 1**
- 1120 The system allows for any number of comments for each transaction. Each comment records the date and user who posted the comment. **Priority 1**
- 1130 Each posted comment triggers a notification to be sent to all users associated with the specific financial account. **Priority 4**
- 1140 The system allows for the sorting of transactions based on any number of valid transaction fields. Typical transaction fields include date ranges, modification date ranges, labels and payee. **Priority 1**
- 1150 The system allows for the searching of transactions based on standard logical search operations (AND, OR, NOT) on any number of valid transaction fields. Typical transaction fields include creation date, modification date ranges, categories and payee. **Priority 1**

**Fig. 1.** VODKA Requirement Specification Snapshot

The service-oriented architecture is designed to maximize reusability and flexibility, and to facilitate communication with external applications. For example, VODKA needs to integrate Drexel authentication systems for user validation, and makes use of external short message system (SMS) and email system to send notifications.

Figure 2 shows the three-tier architectural diagram, in which the components of the system are divided among the presentation layer, business layer, and data layer. Components in the presentation layer interact with the user, validate input data, and display reports. Business layer components provide business logic and data processing capability. Data layer components provide access to the database. Each layer of the system depends only on the layer directly below it, and layers may not depend on layers above them.

This high-level architecture diagram does not provide sufficient basis for task decomposition. For example, it does not show what are the interfaces among the components. According to Baldwin and Clark's modularity theory [3], the interfaces among the components are the *design rules* that decouple otherwise coupled decisions, and are the most important decisions that have to be made beforehand.

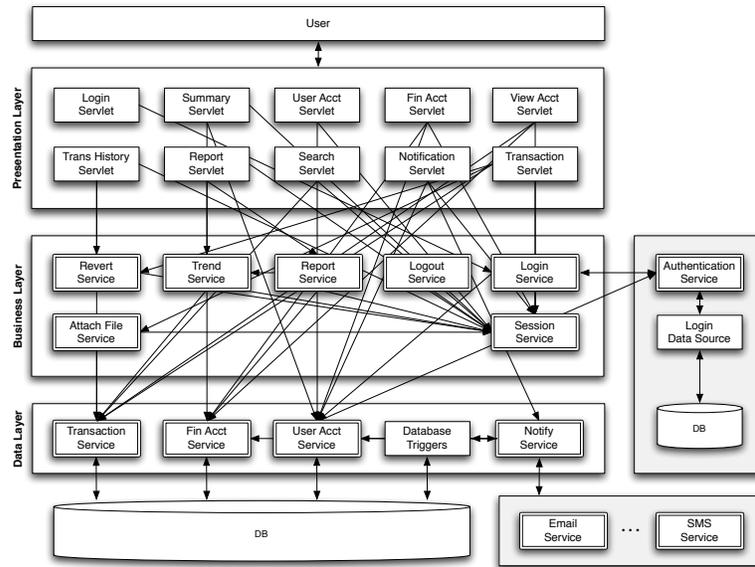
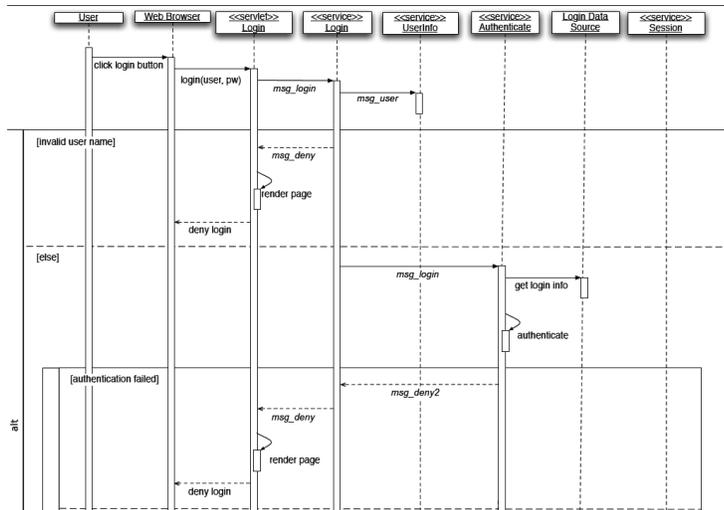


Fig. 2. VODKA Architecture

To model the design in greater detail, the designers model each feature using a UML sequence diagram. There are 24 main features of this systems, and each of them is documented in a subsection of the design document with a description, a sequence diagram, and a note showing which requirements the feature is designed to fulfill. UML sequence diagrams are usually large and span more than one page. Figure 3 shows a snapshot of the UML sequence diagram for the login feature.

We identify the following kinds of design decisions, and model them as variables.

- Each service and servlet in the sequence diagrams is a design dimension that the designers decide to include. The detailed design of each service or servlet can vary. Accordingly, We view each of them as a *variable*. For example, the login service described in section 2.6.2 of the 1st version design document is modeled as: `d_service_login: {v1_2_6_2, other}`.
- The messages passing around the services are also design decisions. This kind of design decisions are particularly important because they are the *interfaces* among difference components. For example, the `msg_login` that goes into service **Authentication** can be viewed as the contract between the internal service **login** and the external service **Authentication**. Determining the format and content of these messages early enables parallel development of these components. In other words, these messages are the architecture *design rules* in Baldwin and Clark’s modularity theory.



**Fig. 3.** Snapshot of VODKA Software Design Documentation

We similarly model these messages as variables. The concrete formats of these messages are not determined and documented, but we know that the concrete decisions may vary. Accordingly, we generally model the possible choices using two values **orig** (short for original) and **other** (some future unelaborated choices). For example, in the following definition, `msg_service_login_requestlostuid: {orig, other}`, the middle part of the name `service_login` means that this is a message decision that have to be made by the login service designer. The content of the message is about lost user id request.

**Testing Decisions** The team also developed an acceptance testing plan documenting test cases according to requirements. Each subsection describes the preconditions, postconditions, and expected actions for each function. The following definition models the testing decision about the account summary function specified in section 6.2.3 of the acceptance testing plan document, version 1: `test_account_summary: {v1_6_2_3, other}`.

**Implementation Decisions** The design is developed as a team, and the implementor of each component will have to make his own decision about how to implement the component. We model the implementation decision of each component with a new variable. For example, `impl_service_login: {v1, other}` models the decision that has to be made by the implementor of the login service for the first release.

**Decision Modeling Summary** At this point, we model the VODKA system with 162 variables using Simon. The model is extensible in the sense that new design components can be added and decisions such as database schema can also be modeled. We could also model other dimensions such as the choices of a web server and other technologies, if needed later.

### 3.2 Model Dependences as Logical Expressions

After modeling these decisions using variables, we now model their dependence relations that span across overall project development stages. In particular, we model the following types dependence relations: (1) dependences between design decisions and requirement decisions; (2) dependences between testing decisions and requirement decisions; (3) dependences between design decisions and implementation decisions; (4) dependences between implementation decisions and design rules. Each of the following line represents one of the four kinds of dependences.

- (1) `d_servlet_login = v1_2_5_2 => spec_security_authen = v1_1390;`
- (2) `test_logging = v1_6_2_1 => spec_user_account_types = v1_0180;`
- (3) `impl_servlet_login = v1 => d_servlet_login = v1_2_5_2;`
- (4) `impl_servlet_login = v1 => msg_service_login_deny = orig;`

The major dependence relation we model is the *assumption* relation. For example, line (1) shows that the design of the login servlet, as specified in section 2.5.5 of the design document, version 1, *assumes* that the security authentication function is as specified in the specification document, version 1, item 1390. The design decision is made corresponding to the requirement. Logically, the binding of the assuming variable implies the assumed binding. This might seem counterintuitive, but there could be other choices for the design of the login servlet that are also consistent with the specification, and we do not want to model an overly constrained design. On the other hand, if the requirement specification changes, the corresponding design decision has to be revisited. Thus the implication arrows are opposite of what one might initially expect.

### 3.3 Model Dominance Relation

Dominance relation is an indispensable design phenomenon but cannot be modeled by logical expressions. For example, in VODKA, the requirements are agreed by the outside stakeholders and thus dominate other decisions. Design rules, such as the message format and interfaces agreed among the designers and implementors, dominate other subordinate implementation decisions. We model these relation as a binary *dominance* relation supplement to the logical constraint networks. For example, the relation includes the pair (`d_servlet_login`, `spec_security_authen`), indicating that the login servlet design should not influence the security specification).

### 3.4 Model Clustering Set

Another phenomenon that cannot be readily expressed by logical expressions is that the same design can be viewed in different ways: in the layered architecture diagram, each layer can be seen as a cluster; each feature crosscuts all the layers, forming a vertical slicing cluster; the client views all the requirements together as a cluster; and all the designs are clustered together into a document. We model this diversity using another data structure called *cluster set*. Each element in the set models one way the system can be clustered, and each cluster is a set variables organized into a tree structure. Note that a cluster is not a *module*. A cluster helps to understand the system in different ways, while a *module* models independent responsibility assignment.

The constraint network, dominance relation, and cluster set constitute a full VODKA augmented constraint network. In the next section, we introduce how Simon automatically decompose the big ACN into *modules* that can be implemented independently, the analysis capabilities enabled, and the issues identified in current design and testing plan.

## 4 Design Analysis

In our previous work [8], we introduced an algorithm to automatically decompose a big ACN model into smaller sub-ACNs, to address the scalability issue caused by the difficulty of constraint solving and the need to enumerate solutions required by the DA. The basic idea is to model the constraint network part of the ACN using a directed graph. In this graph, each node represents a design variable. Two variables are connected if they appear in the same constraint expression. We then cut the edges of the directed graph using the non-trivial dominance relation of the ACN: if A cannot influence B, then the edge pointing B from A will be removed from the graph.

After that, we compute the condensation graph of the resulting graph [2], and put all the variables along the chains ending with the same minimal elements into a sub-ACN, with the relevant subset of constraints, dominance relation and cluster set. As a result, we decompose the big ACN into a set of smaller sub-ACNs that can be solved individually. We also presented the algorithm of integrating analysis results of sub-ACNs into a full solution [8]. More interestingly, we found that each minimal element of the condensation graph represents a function, and all the chains ending with a minimal element contain all the decisions needed to accomplish the function. In other words, each sub-ACN has the potential to be an independent responsibility assignment module. To further evaluate the decomposition algorithm in terms of its ability to decompose a system into independence modules, we use this algorithm to decompose the VODKA ACN into a number of sub-ACNs, analyze each sub-ACN individually, and integrate the results.

The experiment shows that our approach not only generates independent modules, enables traceability analysis and change impact analysis, but also reveals a number of issues in current design and testing plan.

## 4.1 Model Decomposition

Using Simon, the big VODKA ACN is decomposed into 21 smaller sub-ACNs. We notice that the 21 sub-ACNs correspond to the 21 components in the layered architectural diagram shown in Figure 2, although we did not use the digram when we model decisions and their dependencies. Note that there are 24 VODKA features, and a module does not correspond to a feature. Instead, a feature may involve multiple components. For example, the login feature consists of one component from each layer. On the other hand, a component may have multiple features.

Table 1 summarizes these sub-ACN modules, showing the number of variables and the corresponding component of each module. The modules ending with (*UI*) are the components in the representation layer; the modules ending with (*Data*) are the components in the Data layers; and all the other modules are the components in the business layer.

**Table 1.** Decomposed VODKA

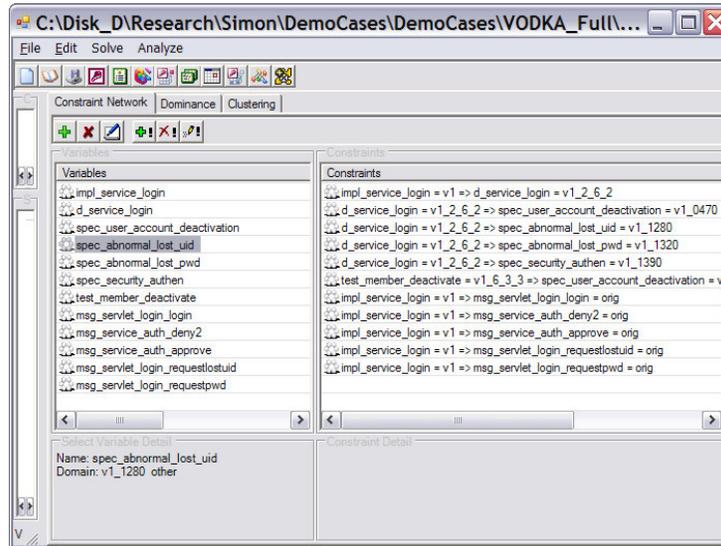
No.	Module	size	No.	Module	size	No.	Module	size
1	attachment	3	8	report	13	15	report (UI)	12
2	fin acct. (Data)	14	9	revert	3	16	search (UI)	4
3	notification (Data)	3	10	session	11	17	summary (UI)	11
4	transaction (Data)	31	11	trend	4	18	transaction (UI)	30
5	user acct. (Data)	17	12	fin acct (UI)	3	19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

After decomposition, we can use Simon to open each sub-ACN module as an independent project. Figure 4 shows one of the sub-ACNs corresponding to the login service in the business layer. Figure 5 shows the corresponding DSM automatically generated by Simon.

Compared with the highly simplified layered architecture diagram, each component is now modeled by an ACN specifying all the decisions needed for the component and the constraints among them. The DSM shows that these are the decisions from different development stages. From the ACN and DSM, we can easily tell that before the login service can be accomplished independently, the designer/implementer has to know the four related requirements (row 0-3), having agreements about two message formats with the authentication service designer (row 4-5), having agreement about three message formats with the login servlet designer (row 6-8), and the implementation should confirm to the design (row 9).

## 4.2 Traceability Analysis

Simon generates all the designs within the design space, from which the user can perform traceability analysis. For example, Figure 6 shows a design for the



**Fig. 4.** Simon Snapshot: the login service sub-ACN

login service. The user can see that the current design and implementation is according to indexed requirement items 1390, 1320, 1280, and 0470, specified in the first version of the requirement specification document.

### 4.3 Change Impact Analysis

If one of the decisions changes, for example, the requirement specified by item 0470, account deactivation, has to be changed, the designer needs to know what other decisions have to be revisited. Using Simon, the designer can perform change impact analysis introduced in our previous paper [7, 8]. For example, if the value of `spec_user_account_deactivation` changes to `other`, Simon will compute that the following decisions need to be revisited: `d_service_data_useracct`, `d_service_login`, `d_servlet_login`, `impl_service_data_useracct`, `impl_service_login`, `impl_servlet_login`, and `test_member_deactivate`. Our recent paper [6] presents a model to analyze software evolution that involves design space expansion. As part of our future work, we will apply the model to VODKA during its feature enhancement stage.

### 4.4 Design Issues Revealed

Some sub-ACNs have relatively large number of variables, such as the 4th sub-ACN with 31 variables, and the 18th sub-ACN with 30 variables. Before we further generate their DSMs, we found that the large size of the module indicates

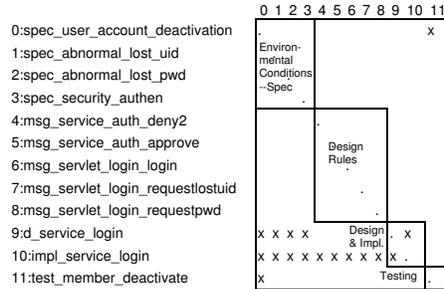


Fig. 5. The logic service DSM generated by Simon

that these components are not very well modularized. For example, the 18th sub-ACN models the transaction servlet in the presentation layer. The sub-ACN indicates that the design of the servlet has to involve many dimensions, such as transaction creation, deletion, modification.

Given the model, the designers realized that this is not a good design because many features (concerns) are tangled together, and it is going to be very difficult to use one webpage as the user interface for so many interactive activities. The big modules suggest the parts that need be reconsidered and redesigned. In other words, the decomposition reflects the quality of the design: if a system is better modularized, the big ACN will be decomposed into larger number of ACNs with smaller sizes.

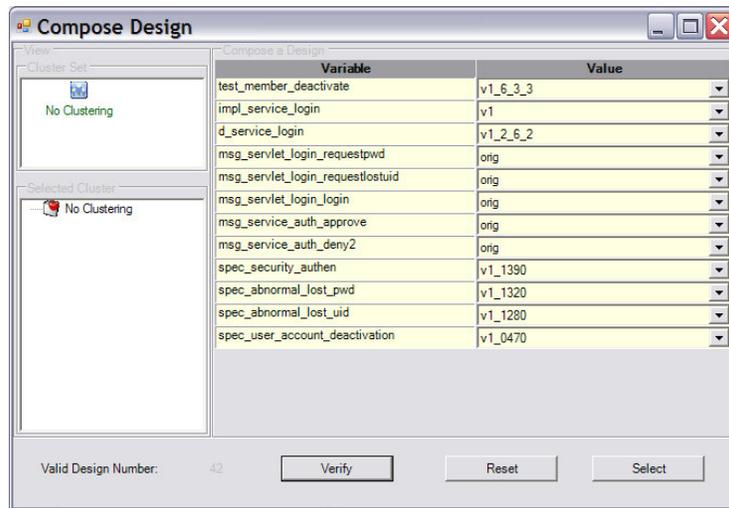
The experiment also reveals other issues in the informal documents. For example, a variable called `d_service_trigger` is never used in any constraint. By checking the design document retrospectively, we found that although this service appears in one of the sequence diagrams, but it is not specified or used in any other models.

#### 4.5 Testing Issues Revealed

The acceptance testing plan specifies a number of test cases. Without our modeling and analysis experiments, it is hard to tell if the testing plan is complete. That is, all required functions should be tested. By examining the sub-ACNs, we found that several sub-ACNs do not have any testing components, such as the notification sub-ACN. We then realized that although notification is an important feature, but is not included in the testing plan. The experiment suggests that we come up with a more complete testing plan.

### 5 Related Work

Many people have explored auto-clustering approaches to decomposing a big dependence model into modules, such as Mancoridis's Bunch tool [15], which is



**Fig. 6.** A design for the login service

based heuristic fitness function. By contrast, our approach uses dominance relations to decompose a big model. Different from the feature-oriented research led by Batory [4], Goguen [11] and Czarnecki [10], we model more general decisions, and view features as one kind of decisions. While their purpose is to synthesize complex software systems from libraries of reusable components, our purpose is to rigorously support modularity analysis and decision-making. Similar to our design space modeling, Lane [13] models the structure of software systems as design spaces. Different from our work, they focus on functional choices. Their notions of rules, similar to our constraints, are formulated to relate choices within a design space. Our modeling approach is more general and formal, supporting automated analysis. Traditional impact analysis research focuses on change issues at program level, as summarized in [1], while our approach works at abstract design level. Our work is different from Kazman's [12] Software Architecture Analysis Method (SAAM) because our approach is based on formal modeling and automatic analysis.

## 6 Conclusion

To address the problem that prevailing design modeling techniques are not designed to generally model, analyze, and decompose decisions that span multiple stages of software lifecycle, we have developed an analyzable ACN design representation and a decomposition algorithm. This paper evaluates our framework by modeling and analyzing the VODKA project. The experiment shows obvious ad-

vantages of our approach: (1) decisions made in various stages are automatically decomposed into independent responsibility assignment modules; (2) traceability and changeability analyses are automated; (3) design issues are identified early and thus can be corrected before making expensive coding investments.

## References

1. R. Arnold and S. Bohner. *Software Change Impact Analysis*. Wiley-IEEE Computer Society Pr, first edition, 1996.
2. S. Baase and A. V. Gelder. *Computer Algorithms: Introduction to Design and Analysis (3rd Edition)*. Addison Wesley, 3rd edition, Nov 1999.
3. C. Y. Baldwin and K. B. Clark. *Design Rules, Vol. 1: The Power of Modularity*. The MIT Press, 2000.
4. D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. The genovoca model of software-system generators. *IEEE Software*, 11(5):89–94, Sept. 1994.
5. Y. Cai. *Modularity in Design: Formal Modeling and Automated Analysis*. PhD thesis, University of Virginia, Aug. 2006.
6. Y. Cai and S. Huynh. An evolution model for software modularity assessment. In *Proceedings of the ICSE Workshop on Assessment of Contemporary Modularization Techniques (ACoM 2007)*, May 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*, Long Beach, California, USA, Nov 2005.
8. Y. Cai and K. Sullivan. Modularity analysis of logical design models. In *21th IEEE/ACM International Conference on Automated Software Engineering*, Tokyo, JAPAN, Sep 2006.
9. S. E. S. Committee. Recommended practice for software requirements specifications. (830), 1998.
10. K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, 1st edition edition, Jun 2000.
11. J. A. Goguen. Reusing and interconnecting software components. *IEEE Computer*, 19(2):16–28, Feb. 1986.
12. R. Kazman, L. J. Bass, M. Webb, and G. D. Abowd. SAAM: A method for analyzing the properties of software architectures. In *International Conference on Software Engineering*, pages 81–90, 1994.
13. T. G. Lane. Studying software architecture through design spaces and rules. Technical Report CMU/SEI-90-TR-18, CMU, 1990.
14. 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.
15. S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, and E. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proceedings of the 6th International Workshop on Program Comprehension (IWPC'98)*, pages 45–52, June 1998.
16. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–8, Dec. 1972.
17. 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.
18. K. Sullivan, W. Griswold, Y. Song, and Y. C. et al. Information hiding interfaces for aspect-oriented design. In *ESEC/FSE '05*, Sept 2005.