

# Modularity Analysis of Logical Design Models

Yuanfang Cai

Dept. of Computer Science  
University of Virginia  
Charlottesville, VA, 22904-4740 USA  
yc7a@cs.virginia.edu

Kevin J. Sullivan

Dept. of Computer Science  
University of Virginia  
Charlottesville, VA, 22904-4740 USA  
sullivan@cs.virginia.edu

## Abstract

*Traditional design representations are inadequate for generalized reasoning about modularity in design and its technical and economic implications. We have developed an architectural modeling and analysis approach, and automated tool support, for improved reasoning in these terms. However, the complexity of constraint satisfaction limited the size of models that we could analyze. The contribution of this paper is a more scalable approach. We exploit the dominance relations in our models to guide a divide-and-conquer algorithm, which we have implemented it in our Simon tool. We evaluate its performance in case studies. The approach reduced the time needed to analyze small but representative models from hours to seconds. This work appears to make our modeling and analysis approach practical for research on the evolvability and economic properties of software design architectures.*

## 1. Introduction

Reasoning about modularity properties of software design architectures and their associated technical and economic implications is critical to high-consequence decision-making in software design. However, such reasoning remains difficult and largely intuitive today. One problem is that we lack tractably analyzable high-level design representations that both convey design architectures and enable designers to reason precisely about design structures and their key technical and economic implications.

Baldwin and Clark contributed an influential but informal theory of modularity, centered on a representation called the design structure matrix (DSM). They linked DSMs to economic value based on a theory of real options [2]. Sullivan et al. [20] showed that this model could be extended to make precise Parnas's notion of information hiding [17], and that the economic predications of the model agreed with Parnas's conclusions. Sullivan et al. [21]

later showed that the ideas could account for aspect-oriented concepts of modularity, and in particular could lead to new forms of interfaces for improved modularity in that area.

Our prior work and that of others, including Lopes et al. [13] and Sangal et al. [18], have showed that the ideas of Baldwin and Clark promise to enable new approaches to structural and economic analysis of software design architectures. However, in reflecting on this work, we have also been troubled by the lack of a clear semantics for, and by observed difficulties in producing and validating, DSM models.

We thus developed a formal modeling framework, supported by our tool, *Simon*, that helps to clarify Baldwin and Clark's theory in the formal setting of finite-domain constraint networks [4, 3]. Our framework is based on the use of what we call *augmented constraint networks* (ACNs) to model design architectures. *Simon* enables construction and analysis of ACNs. Given an ACN, *Simon* derives a state-machine-based model that we call a *design automaton* (DA) that supports a range of economic and evolvability analyses. For example, from a DA, *Simon* can derive what we call a *pair-wise dependence relation* (PWDR) on design variables, and from a PWDR, a DSM, enabling direct use of Baldwin and Clark's net option value (NOV) analysis.

As with many formal techniques, such as model checking, the complexity of constraint satisfaction limits the size of models that can actually be analyzed. Our DA model also requires an explicit representation of the set of all satisfying solutions, which in general grows exponentially in the number of variables in a model.

This paper addresses these scalability problems. We make use of what we call a *dominance* relation in an ACN model to split an ACN at natural breaking points into sub-ACNs. We solve each sub-ACN separately and then integrate partial results on demand. The gain comes both from the reduction in the sizes of the SAT problems and from avoiding the need to explicitly generate the full DA. Our case studies suggest that this approach can reduce the time required to analyze modest design models from hours to

seconds.

We illustrate our approach using Parnas’s canonical *Key Word In Context (KWIC)* example. To evaluate our method, we applied it in case studies of HyperCast, a peer-to-peer network [12], as studied by Sullivan et al. [21], and the WineryLocator application studied by Lopes et al. [13]. The cited works presented DSM models of the respective systems. We used Simon to model them with ACNs, but we were unable to get results quickly enough. Our algorithm reduced the times from hours to seconds. It also appears to decompose ACNs into meaningful design modules.

The rest of this paper is organized as follows. Section 2 introduces the background of this work. Section 3 presents our algorithm for modularizing ACNs. Section 4 presents the algorithms of composing the results obtained by analysis of the individual modules. Sections 5 and 6 present our case studies and evaluation results. Section 7 presents related work. Section 8 discusses related issues and concludes.

## 2. Background

In his seminal paper [17], Parnas presented a comparative analysis of the *changeability* of designs produced by two major modularization approaches, based on their ability to accommodate changes in key environmental conditions (specifications of input format, input size, core size and alphabetizing policy). Current design representations such as the Unified Modeling Language (UML) and most Architectural Description Languages (ADLs) [8] are neither designed to represent such external conditions nor to support precise analysis of their influences on design evolution.

### 2.1 Emerging New Approach

Sullivan et al. [20] showed that DSMs could represent such external conditions, and that Baldwin and Clark’s model of modularity in design, so extended, could account for Parnas’s criterion of information hiding. In particular, information hiding is seen to mean that special design variables, the design rules that decouple modules, are invariant under changes in environment variables. We present both our modeling approach and our new algorithm in terms of the example developed in that earlier work, as reformulated in terms of ACNs.

DSMs present, in a matrix form, the pair-wise dependence relations on design decisions and the corresponding development and evolution process. Figure 1 shows a KWIC DSM model. The rows and columns of a DSM are labeled with design variables, uniformly representing the environment conditions or design dimensions for which the designers must make design decisions. Marked cells

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1:envr_input_format																				
2:envr_input_size			x																	
3:envr_core_size		x																		
4:envr_alph_policy																				
5:linestorage_ADT																				
6:input_ADT																				
7:circ_ADT																				
8:alph_ADT																				
9:output_ADT																				
10:master_ADT																				
11:linestorage_ds		x	x																	
12:linestorage_impl		x																		
13:input_impl		x																		
14:circ_ds		x	x																	
15:circ_impl																				
16:alph_ds																				
17:alph_impl																				
18:output_format																				
19:output_impl																				
20:master_impl																				

Figure 1. KWIC IH DSM

model the dependence relation on design decisions. For example, the cell in row 11 (*line\_storage\_ds*) and column 2 (*envr\_input\_size*) indicates that how to store data depends on how large the input size is.

Software architecture can be revealed by ordering and clustering the rows and columns of a DSM. In Figure 1, variables starting with “envr\_” are clustered into an *Environment* module, representing external forces of change on the design. The lower right part represents the *Design* comprising abstract data type interfaces, data structures and sub-program implementations. Variables ending with “\_ADT”, representing the abstract interfaces in Parnas’s KWIC information hiding design, are clustered into a *Design Rule* module. The marks below the environment conditions and to the left of the design indicate that the design is subject to environment changes. Similarly, within the design, the functional modules, comprising data structures (ending with “\_ds”) and implementations (ending with “\_impl”), depend on the abstract data types. Figure 1 shows that there are six modules to implement: line storage, input, circular shift, alphabetizing, output, and master control. The absence of off-diagonal marks indicates no dependences between the function modules, revealing a modular structure. Figure 1 shows that once the abstract interfaces are determined, these modules can be implemented independently in parallel.

According to Baldwin and Clark [2], design rules are defined as architectural design decisions that *dominate* other design decisions, decouple otherwise coupled design decisions, and remain stable. One of the central operations in a design activity is to create design rules so that modules come to depend on interfaces, but are independent of each other, producing a *modular* design. The *dominance* relation among design decisions is an indispensable software design structural property. In addition to *design rules*, the fact that the environment conditions are usually out of the

designer’s control is another type of dominance relation. A DSM represents the *dominance* relation by asymmetric dependences. The absence of dependences in the upper right part of Figure 1 indicates that these environment conditions are not controlled by the designers, and that the abstract data types are design rules. We see that DSM modeling reveals Parnas’s information hiding criterion visually: a design exhibits information hiding modularity if the design rules are invariant with respect to changes in environment variables. In such cases, such external changes are accommodated solely by changes within the hidden modules of a design. Figure 1 shows Parnas’s KWIC information hiding (IH) design modeled using a DSM.

Despite its strengths, DSM modeling is informal and not sufficiently expressive to enable rigorous and precise evolvability or economic analysis. First, it represents design dimensions but not the choices available in each dimension, so one cannot analyze the precise impacts of detailed decision changes in given design dimensions. For example, Parnas mentioned in his paper that input data can be stored in different ways: packing four characters into one word, no packing, or disk storage. There could also be other ways to store data. However, in a DSM, we can only model that there is a dimension called “*linestorage\_ds*”. The concrete conditions are not represented explicitly. Second, there are usually multiple ways to accommodate a change, but a DSM model does not represent them separately. Third, the semantics of markings in DSMs are not clear, and, in our experience, it can be hard to decide which marks to include or not. Our recent work [4] has shown errors in published manual models. Many of the errors are due to the difficulty of seeing transitive relations among dependencies. We have found that marking dependencies in large DSMs to be error-prone and time-consuming.

## 2.2 Formal Modeling and Automated Analysis

Our previous paper [4] presents a formal analyzable design representation to address these problems, capturing and clarifying the essence of Baldwin and Clark’s theory, enabling the application of essentially all existing DSM-based analysis techniques, including net options value analysis, to DSMs having precise dependence semantics, and enabling a number of additional architecture analysis techniques.

We take finite-domain *constraint networks* (CNs) [15] as the core of our design representation. Each design dimension or external condition is represented by a *decision variable*. A choice within a dimension is represented by a binding of a *value* from a finite set of possible values—its *domain*—to a variable. For example, the domain of the variable `linestorage_ds` is `{core4, core0, disk, other}`. Dependences among decisions are expressed as logical *constraints*. We aug-

mented a logical constraint network representation with a binary *dominance* relation to model the asymmetric dependence relation among design decisions. For example, `(linestorage_ds, envr_input_size)` and `(linestorage_ds, input_ADT)` are the elements of the KWIC dominance relation, indicating that how to store data should not influence the input size and the abstract data type. Another extension is a set of *clustering* data structures to allow the analyst to explore the implications of different approaches to aggregating decisions into modules. We refer to a constraint network augmented with a *dominance* relation and a *clustering set* as an *augmented constraint network* (ACN). For the sake of space, Figure 2 only shows the constraints modeling Parnas’s IH design.

```

1: linestorage_impl = orig => linestorage_ADT = orig
   && linestorage_ds = core4;
2: linestorage_ds = core4 => envr_input_size = medium
   || envr_input_size = small;
3: linestorage_ds = core0 => envr_input_size = small
   && envr_core_size = large;
4: linestorage_ds = disk => envr_input_size = large;
5: circ_ds = copy => envr_input_size = small || envr_core_size = large;
6: circ_impl = orig => circ_ADT = orig && circ_ds = index
   && linestorage_ADT = orig;
7: input_impl = orig => input_ADT = orig;
8: alph_impl = orig => alph_ADT = orig && alph_ds = orig;
9: output_impl = orig => output_ADT = orig && output_ds = orig;
10: master_impl = orig => master_ADT = orig && linestorage_ADT = orig
    && input_ADT = orig && circ_ADT = orig
    && alph_ADT = orig && output_ADT = orig;
11: alph_impl = orig => circ_ADT = orig && linestorage_ADT = orig;
12: input_impl = orig => linestorage_ADT = orig;
13: output_impl = orig => linestorage_ADT = orig && alph_ADT = orig;
14: alph_ds = orig => envr_alph_policy = once;
15: input_impl = orig => envr_input_format = orig;
16: alph_impl = orig => envr_alph_policy = once;

```

Figure 2. KWIC IH Constraint Network

Simon takes an ACN as input, and derives a nondeterministic automaton to represent the change dynamics within a modeled design space, which we call a *design automaton* (DA). The states of a DA represent design configurations that satisfy all constraints. Transitions model changes in designs, driven and labeled by changes to individual design decisions. The destination state for a given starting state and change differs from the initial state in a way that is *minimally* sufficient to restore consistency. We have also introduced an approach to design impact analysis (DIA) at the architectural level, enabled by DA [4], which aims to answer the following question: given a software design, what are all the ways to compensate for an anticipated sequence of individual changes? The question can be formulated as a mapping from a DA, an assignment modeling the current design, and a sequence of variable-value pairs that model changes, to a set of sequences of consistent design states modeling the nondeterministically feasible evolution paths

for the given sequence of changes. The solution is straightforward based on the DA model: we find the paths that start from the initial design and go along the edges labeled with specified changes. Each path represents one way to compensate for the given changes. The destination states are the possible new design states accommodating the given sequence changes [4].

The DA model provides a precise definition of what it means for one variable to depend on another: we define two design variables to be *pair-wise dependent* if, for some design state, there is some change to the first variable for which the second must change in at least one of the minimal compensating state changes. Accordingly, Simon derives a *pair-wise dependence relation* (PWDR) from a DA, combines it with a selected clustering, and generates a DSM. Figure 1 is the KWIC DSM automatically generated by Simon using the ACN for which Figure 2 presents the constraint network. As a result, we can obtain all the analysis capabilities of a DSM from an abstract logical model. We have shown [4] that the automatically derived DSMs include easily overlooked transitive dependences, the source of modeling errors in several previous studies using DSMs.

However, as with many formal analysis techniques, the difficulty of constraint satisfaction limits the size of models that can be analyzed in practice. Our DA model is even more demanding, because it requires the explicit representation of the entire space of satisfying solutions. The number of the solutions increases exponentially in the number of variables involved (in general), and it is impractical to represent DAs explicitly in cases where the state spaces are very large. In the following sections, we present our method to split constraint networks using the *dominance* relations at natural breaking points. We then solve the sub-models and integrate partial results, but only as needed, to produce the desired answer more efficiently.

### 3. ACN Splitting

To provide a full picture of how this approach works, we consider part of the KWIC information hiding model involving the first 6 constraints shown in Figure 2. There are 8 design variables involved in these constraints. The dominance relation of the example dictates the following: (1) variables starting with “\_envr” (environment variables) should not be influenced by any other variables; (2) variables ending with “\_ADT” (design rules) should not be influenced by variables ending with “\_ds” or “\_impl”. Our splitting approach takes the following steps:

First, we construct a graph depicting how these variables are syntactically connected. To construct such a graph, we first change the involved constraints into a conjunction normal form (CNF). Without loss of generality, we assume that each variable is involved in at least one clause. After that,

we model each clause using a directed complete subgraph: each variable is a node, each node connects to every other node, and their values are ignored. As a result, the whole CNF transforms into a directed graph:  $G_{cnf} = \langle V, E \rangle$ .  $V$  is variable set of the ACN. We assume that this graph is at least weakly connected. Otherwise, we consider each sub-graph separately. This graph models the most conservative dependence relation among variables: if two variables appear in the same clause, they depend on each other syntactically. As a result, in Figure 3, every variable connects with every other variable directly or indirectly.

Second, we remove edges according to the *dominance* relation. If a variable pair  $(v_i, v_j) \in dominance$ , we remove the edge  $\langle v_i, v_j \rangle$  from the  $G_{cnf}$ . We call the resulting graph  $G = \langle V, E \rangle$ . If  $G$  is not weakly connected, we consider each sub-graph separately. In Figure 3, the dotted lines labeled X are the edges to be excluded.

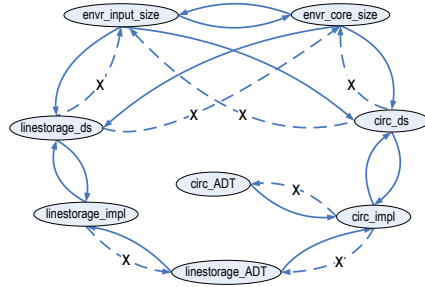


Figure 3. Partial KWIC CNF graph

Third, we construct the corresponding condensation graph. We use M. Sharir’s algorithm [1] to find strongly-connected components of  $G$  and construct its condensation graph:  $G^* = \langle V^*, E^* \rangle$ . Figure 4 shows the condensation graph of Figure 3, in which  $V^* = \{V0, V1, V2, V3, V4\}$ . Each node of  $G^*$  represents a strongly-connected component of  $G$ , comprising a set of variables.  $G^*$  is a *directed acyclic graph* (DAG) [1], containing a partial order of  $V^*$ .

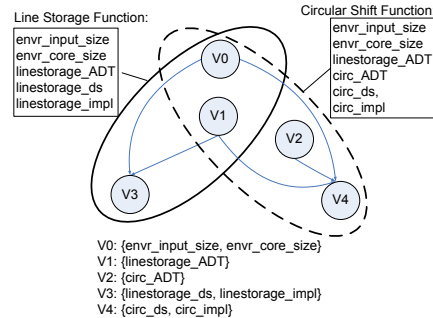


Figure 4. Condensation Graph

Fourth, we construct sub-ACNs. The number of sub-ACNs equals to the number of minimal elements of  $G^*$ . Figure 4 has two minimal elements:  $V3$  and  $V4$ , so we can construct two sub-ACNs in the following way:

1. Construct variable set. For each minimal element, the union of all the nodes, each being a set of variables, of  $G^*$  that on the chains ending with the element is the variable set of a sub-ACN. According to Figure 4, the variable set of the first sub-ACN is the union of  $V0$ ,  $V1$ , and  $V3$ . We observe that they are all the elements needed for the line storage function. The union of  $V0$ ,  $V1$ ,  $V2$ , and  $V4$  is the variable set of the second sub-ACN modeling the circular shift function.
2. Construct the constraint set for each sub-ACN. If the variable set of a sub-ACN contains all the participating variables of a CNF clause, we put the clause into the constraint set of the sub-ACN. It is possible that there are clauses that do not belong to any sub-ACN. In this case, we consider the graph,  $G_{left}$ , made from the complete graphs of these clauses. Each connected component of  $G_{left}$  forms a new sub-ACN: the corresponding clauses form its constraint set; all the variables involved in these clauses form its variable set.
3. Construct the *dominance* relation for each sub-ACN. From the *dominance* relation of whole ACN, each sub-ACN selects the subset that only involves its own variables.

We have observed that in practice this method tends to group variables into cohesive, sparsely overlapping sets that correspond to key features of the design. In a sense, our decomposition approach realizes a kind of *architectural slicing*. From the running example, we obtain two sub-ACNs corresponding to the line storage function and the circular shift function respectively. We decompose both the KWIC sequential and information hiding (IH) ACNs introduced in our previous work [4], compare the decomposed sub-models, and summarize them in Table 1. We observe that the information hiding design has one more sub-ACN, and that most information hiding sub-ACNs are smaller than the sequential sub-ACNs. From the IH sub-models, it is easy to tell that each sub-model corresponds to a main function: line storage, input, circular shift, alphabetizing, output, and master control. However, after decomposing the sequential ACN, it is harder to see clearly what function each sub-ACN models. This observation suggests that perhaps the design is not as well organized as a system of individually coherent parts, and even that our analysis approach might be able to shed early lights on such design problems.

Without decomposition, the KWIC sequential ACN with 18 variables took Alloy [10]—the SAT solver Simon employs—about an hour on a Pentium 1.5 GHz, 512 MB RAM PC to find the 12018 solutions and then 11 minutes

**Table 1. KWIC ACNs Decomposed**

	KWIC IH			KWIC Sequential		
	Size	CN (sec)	DA (sec)	Size	CN (sec)	DA (sec)
1	6	14	< 1	9	6	< 1
2	6	18	< 1	8	22	< 1
3	4	6	< 1	8	17	< 1
4	5	9	< 1	9	10	< 1
5	7	20	< 1	6	68	< 1
6	5	8	< 1			

to compute the DA and DSM. After decomposition, it takes about 1 minute in total to get the integrated result. The full IH model with 20 variables took about three hours to find 34907 solutions, and the DA and DSM computation took another 2 hours 13 minutes. After decomposition, getting integrated results using the approach introduced in the next section takes half a minute in total.

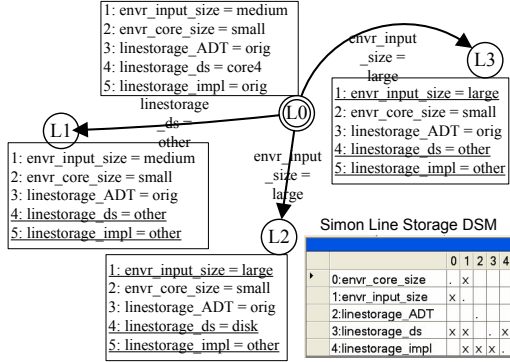
Alloy was not designed for exhaustive enumeration, so some of the observed inefficiency is perhaps due to this mismatch between tool and application. However, after decomposition, each sub-ACN is still solved by Alloy. The performance gains come from the fact that Simon now invokes multiple solvers, each solving a much smaller model, and then integrates the results quickly.

## 4. Integrating Analysis Results

As we have already explained, the *design automaton* (DA) is the key model enabling both the derivation of the PWDR model and design impact analysis. However, integrating these sub-DAs into a full DA would again be costly. Fortunately, for some of the analyses that we are interested in, it is not necessary to depend on an integrated full DA. Instead, analyses can be done on each sub-ACN, and the results can be integrated on demand to provide the same solution as would have been obtained by whole-DA analysis, but without the need, at least in cases such as we have analyzed, to build and represent the whole DA. In particular, we can compute design structure matrices and analyze design change impact in this way.

### 4.1. Integrating Design Impact Analysis

We consider the basic design impact analysis question: given an original design, what are all the ways to compensate for a design decision change (or an environment condition change)? Instead of modeling the original design as a solution to the ACN, deriving the full DA of the ACN, and finding the solution, as introduced in our previous paper [4], this section presents a method to find sub-solutions from each sub-ACN and sub-DA, and integrate them into a full solution. We continue to use the partial KWIC IH ACN



\*Note: the underlined variables are changed due to the transition.

Figure 5. Linstorage sub-Models

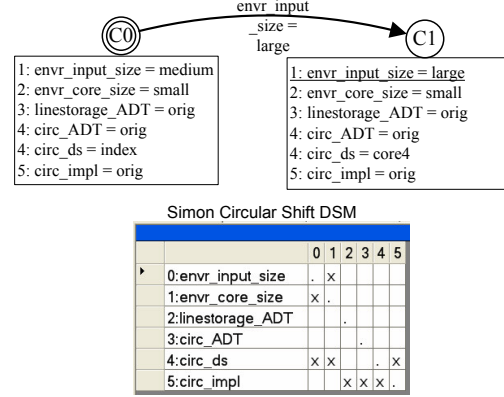


Figure 6. CircularShift sub-Models

in the previous section as an example, and suppose that the original design is as follows:

- 1: envr\_input\_size = medium
- 2: envr\_core\_size = small
- 3: linestorage\_ADT = orig
- 4: linestorage\_ds = core4
- 5: linestorage\_impl = orig
- 6: circ\_ADT = orig
- 7: circ\_ds = index
- 8: circ\_impl = orig

We have shown that this partial ACN can be split into two sub-ACNs. For the designated starting design, we first find the corresponding start states from each sub-DA, in this example, the state  $L0$  in Figure 5 and state  $C0$  in Figure 6. We call  $L0$  and  $C0$  *compatible* states because their shared variables have the same values. Given a changing variable, we distinguish the following two cases.

If the changing variable is local, that is, no other sub-ACNs involve this variable, then the design impact analysis can be done locally using the sub-DA, such as the *linestorage\_ds* in Figure 5: changing its value to other leads state  $L0$  to  $L1$ .

If the changing variable is shared among sub-ACNs, e.g., *envr\_input\_size* appears both in Figure 5 and Figure 6, we just need to integrate the results as follows.

(1) Find the destination states from each sub-DA labeled with this change: in Figure 5, changing *envr\_input\_size* to *large* reaches  $L2$  and  $L3$ ; in Figure 6, the same change leads to  $C1$ .

(2) Compute the cross product of these two sets of states, that is,  $\{L2, L3\} \times \{C1\}$ . In this procedure, a state in one sub-DA is only unified with another state in another sub-DA when these two states are compatible. As a result,  $L2 \cup C1$  and  $L3 \cup C1$  are the two designs that we are looking for.

In other words, if the full DA for the full ACN had been generated, the design impact analysis for the same original design under the same change would have led to the design states that are identical to the integrated destination states.

## 4.2 Integrating Coupling Structures

It is possible that a sub-DA solution satisfies its own constraints, but makes the full constraint network inconsistent. We call such a solution of a sub-DA an *incompatible solution*. In order to compose the pair-wise dependence relation (PWDR) of the full ACN from the derived sub-PWDRs, we first need to remove these incompatible states from each sub-DA. After that, we derive sub-PWDRs from sub-DAs and union the sub-PWDRs together to get the PWDR for the full ACN.

This method does not reduce the big-O complexity of deriving a PWDR from a constraint network, which is NP-complete. In fact, the operation of removing incompatible states has exponential complexity. The essence of our method is to provide a balance between two extremes: on one extreme, a large ACN is solved as a whole; on the other extreme, each clause of a CNF expression can be seen as an individual ACN that can be solved independently. However, in order to integrate the sub-PWDRs derived from these sub-ACNs, comparing each solution of each sub-ACN with every other solution in every other sub-ACN would be again time-consuming. We have formalized the sub-models and proved the correctness of the decomposition and integration [3], but we are not able to present the formalization in this paper for the sake of space.

Our method decomposes an ACN so that each sub-ACN only needs to be compared with other sub-ACNs that share variables with it. For example, the two KWIC sub-ACNs we present have three shared variables. The comparison and incompatible state removing operation are executed along

with the sub-DA generation procedure. People have explored many methods to decompose and cluster a constraint network. As we discuss in section 7, our method is orthogonal to these methods, and combining them may further improve the performance. Figure 5 and Figure 6 also present two snapshots of the DSMs Simon derives from the sub-DAs. These two sub-DSMs are parts of the full DSM shown in Figure 1. We have used Simon to combine full DSMs for both the information hiding and sequential designs, which are exactly the same as the DSMs we generated from full ACNs, as presented in our previous paper [4].

## 5. A Web Application

Lopes et al. used DSMs to model and compare object-oriented and aspect-oriented (AO) designs for *WineryLocator*, a web application [13]. Their purpose, analogous to that of Sullivan et al. [20], was to model the value of modularity, in this case, with a focus on the benefits of aspect-oriented modularity. In this section, we first briefly present how these designs are modeled, and then introduce how large ACNs are split into a number of smaller ones that are solved individually, and present the integrated results. Finally, we compare the discrepancies between our derived DSMs and the manual models in their paper. The differences revealed several problems with their DSMs, suggesting again that informal DSM modeling of abstract design architectures can be error-prone.

### 5.1 ACN Modeling

We first construct the ACN model according to the application description. To locate wineries and get directions using the application, the user first inputs either an approximate or exact address, by which the application locates the exact address as the valid starting point—a function called *startWineryFind*. After that, the user can select preferences for the wineries—the *searchWinery* function. Given a starting point and the preferences, the application generates a route for a tour consisting of all the wineries that match the preferences—the *tour* function. The application also presents driving directions upon user request—the *directions* function.

The application depends on *MapPoint* as the main address and routing service. Lopes et al. developed a local service *WineryFind* to find wineries matching criteria. The interfaces provided by these services are called *MapPointDesignRules* and *WineryFindDesignRules*. In order to authenticate *MapPoint* service requests, each authentication function has to implement a Java *Servlet* interface *HttpSessionListener*, and uses *ApacheAXIS* to insert authentication parameters to *MapPoint*.

There are three supporting functions: the function to get addresses from *MapPoint—AddressLocator*; to get routes from *MapPoint—RouteMapHandler*; and to get wineries from the local service—*WineryFinder*. Since *AddressLocator* and *RouteMapHandler* access the *MapPoint* service, they have to be authenticated. They do so by making two new classes, *AuthAddressLocator* and *AuthRouteMapHandler*, which inherit from *AddressLocator* and *RouteMapHandler* respectively. The authentication is taken care of by these subclasses. All the web service function calls have to be logged by *WebServiceLogger*.

To model this system as an ACN, we add “\_interface” and “\_impl” to the end of each function name to represent the fact that, in an OO design, each function leads to an interface and an implementation. For example, *AddressLocator\_interface* and *AddressLocator\_impl* are the two variables addressing the *AddressLocator* function. We call this design *WineryLocator\_OO*. The constraints among these variables include the following categories: (1) the relations among service functions; (2) supporting functions require that relative services are available; (3) implementations depend on interfaces; (4) user functions depend on supporting function interfaces and other user function interfaces they use; and (5) there are also object-oriented constraints: for example, *AuthAddressLocator* inherits from *AddressLocator*. For the sake of space, we are not able to show the ACN model.

In order to improve the *WineryLocator\_OO* design, Lopes et al. introduced five interfaces to isolate the effects of *MapPoint* as much as possible. Following how they name these interfaces, we model them as the following design variables: (1) *startAddress\_Address*—the starting location the user provides and selects; (2) *matches\_Address*—the data structure storing the set of matched addresses; (3) *WinerySearchOption*—the data structure storing the preferences; (4) *Tour*—the tour representation; and (5) *MapOptions*—standard map options. We call this design *WineryLocator\_DR*. The constraints are adjusted so that some design variables now assume these design rules.

The authors then presented an AO design where logging and authentication functions are implemented using aspects. Modeling the AO design is similar to modeling the OO design. The aspects are also modeled simply as decision variables: in this case, *aop\_Authentication* and *aop\_WebServiceLogging*. The constraints in the AO design change a little: these aspect variables now assume the implementations of other functions. The other functions no longer need to be aware of these two functions. In essence, modeling AO and OO designs are similar. In both ACN models, logging and authentications are just variables.

In modeling the *dominance* relations in these designs, we assume that nothing could affect these third-party services and interfaces; and implementations should not affect

**Table 2. WineryLocator ACNs Decomposed**

	OO design			DR design			AO design		
	N	CN	DA	N	CN	DA	N	CN	DA
1	5	4	<1	8	20	<1	7	11	<1
2	6	6	<1	6	7	<1	6	8	<1
3	6	5	<1	6	6	<1	7	5	<1
4	6	8	<1	5	6	<1	5	5	<1
5	5	4	<1	8	20	<1	7	11	<1
6	11	110	<1	8	36	2	7	18	<1
7	8	20	<1	8	27	<1	7	28	<1
8	9	28	<1	9	51	2	8	40	2
9	2	2	<1	2	3	<1	7	19	<1
10	6	7	<1	8	21	<1			

the specified design rules and interfaces. In the AO design, the authentication and logging aspects shouldn't affect the functions they advise.

### 5.2 Modular Analysis Results

Without decomposition, it took Simon a whole day to solve these constraint networks. DA generation took several days. We simply were not able to generate DSMs in a reasonable time without decomposition. Table 2 lists the number of variables (N), the constraint solving time (CN), and the DA generation time (DA) for each sub-ACN of each design in the unit of seconds. All the experiment data are obtained using a Pentium 1.5 GHz, 512 MB RAM PC. Simon is a C# program, and its output data are written in text files. Since Simon solves sub-ACNs separately, the constraint solving bottleneck depends on the largest sub-ACN to solve.

The *WineryLocator\_OO* ACN with 27 variables is decomposed into 10 sub-ACNs in which the 6th sub-ACN with 11 variables takes about 2 minutes to find all its solutions. After that, DAs and DSMs are generated within a second. The *WineryLocator\_DR* ACN with 32 variables is also decomposed into 10 sub-ACNs, in which the largest sub-ACN took about 1 minute to solve. The *WineryLocator\_AO* ACN with 29 variables is decomposed into 9 sub-ACNs, in which the largest sub-ACN took about 40 seconds to solve. We observe that the third design has one fewer sub-ACN, but its largest sub-ACN is smaller than the largest one of the second design, and the performance is a little better.

### 5.3 Comparative Results

In order to compare the DSMs that Lopes et al. developed informally and our DSMs, derived algorithmically from precise ACNs, we clustered and named the variables of our DSMs to correspond to those of Lopes et al. For example, we cluster *tour\_sig* and *tour\_impl* together as

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	
0:MapPoint	.																							
1:WineryFind	.																							
2:ApacheAXIS	.																							
3:Servlet	.																							
4:HttpSessionBindingListener	.																							
5:MapPointDesignRules	x																							
6:WineryFindDesignRules	x	x																						
7:startAddress_Address																								
8:matches_Address																								
9:WinerySearchOption																								
10:Tour																								
11:MapOperation																								
12:WebServicesLogger																								
13:AddressLocator	x																							
14:AuthAddressLocator	x	x	x																					
15:WineryFinder	x	x	x																					
16:RouteMapHandler	x																							
17:AuthRouteMapHandler	x	x	x																					
18:startWineryFind	x																							
19:searchWinery																								
20:tour																								
21:directions																								
22:web_xml	x	x	x																					

**Figure 7. WineryLocator DR DSM**

a module named *tour*, mapping to the variable with the same name in their DSMs. Figure 7 shows a clustered DR DSM for comparison with their manually-constructed DSMs. Tracing down the differences exposed several interesting issues, showing the advantages of our formal model and automatic tool.

First, our derived DSMs reveal many indirect dependencies not shown in their manual ones. For example, they chose *MapPoint* as their major library, which influences many other decisions. However, in their DSMs, only one module depends on it. Although a higher order matrix might reveal indirect dependences, these indirect dependences are not accounted for in Baldwin and Clark's value model, which they are using [2]. By contrast, our DSMs, derived from a declarative model of design dependences, fully represent all of the relevant transitive dependences that have to be accounted for in order to apply Baldwin and Clark's value model.

Second, the dependence definition in Lopes et al.'s manual DSM modeling is ambiguous, making the manually-constructed DSMs hard to understand. We take three design dimensions, *startWineryFind*, *AddressLocator* and *AuthAddressLocator*, as examples. The first is a function making use of the service provided by the second to locate addresses. The third inherits from the second and extends it with authentication functions. While our derived DSMs show that the first depends on the other two, their manual DSMs indicate only a dependence of *startWineryFind* on *AuthAddressLocator*, but not on *AddressLocator* (row 18, column 13), despite the fact that *AddressLocator* interface changes affect the *startWineryFind* function directly.

We understand the reason for this discrepancy after dis-

Discussing it with the authors to learn exactly how the system is implemented. The dependence between *startWineryFind* and *AuthAddressLocator* is because of *usage*: the former is a *jsp* page using the latter as a Javabean. Since *startWineryFind* doesn't refer to *AddressLocator* directly, the authors did not mark them as dependent. The *usage* and *inherits* relations are different, so using transitive operations to find this missing dependence doesn't seem to be proper. By contrast, our framework provides an exact semantics of *dependence*: some minimal compensation for a change in one design decision involves revisitation and revision of the other. Using this definition, the missing dependences are discovered directly.

Finally, the authors modeled third-party services, such as *MapPoint*, as environment parameters. However, we understand the environment conditions as those that are likely to change and drive software evolutions. For example, the user interface could be either web-based or GUI application based on Java Swing. The authors mentioned this as a possible change, but didn't model and analyze it. These discrepancies imply potential problems in their later quantitative analysis, but we don't go further in this dimension.

## 6. HyperCast

We used HyperCast [12], an independently developed scalable, self-organizing overlay system developed using Java, as a subject in our work applying the design rule concepts of Baldwin and Clark to meet the need for a new kind of crosscutting interface to decouple aspects from the code they advise [21]. We developed two methods to modularize the scattered logging code in the original object-oriented (OO) design. One method is to obviously add logging aspects, as aspect programs often do, resulting an *obliviousness* design. Another method is to insert interfaces that carry design rules to decouple aspects from the code they advise, which we call the *design rule* (DR) design. We used DSMs to represent these three designs, and to quantitatively evaluate the resulting design structures, following the methods of Baldwin and Clark as adapted to software by Sullivan et al. [20]. However, we had to spend a great deal of time producing and correcting DSM models for this study. In order to ease the DSM-based coupling structure analysis and enable design impact analysis, we modeled the three designs as ACNs and tried to use Simon to analyze them as a whole. Although the modeling exercise was profitable, we were not able to obtain analysis results within a reasonable amount of time, until we improved Simon with the decomposition capability.

For the sake of space, we elide the ACN modeling of these designs and report the decomposition results directly. The OO design is decomposed into 6 sub-ACNs, each concentrates on one major task and has several crosscutting

logging-related variables and constraints. The oblivious ACN is decomposed into 5 sub-ACNs. In this case, each sub-ACN is related to one kind of logging. As a result, we got two large sub-ACNs with 17 variables. Taking a closer look at these large sub-ACNs, we found that a lot of variables are replicated in both sub-ACNs, due to the "oblivious" exception logging aspects. These aspects actually depend on the implementations of many other functions. As a result, our decomposition algorithm aggregates these aspects with all the functions that they are advising. The DR ACN is decomposed into 10 sub-ACNs. Each sub-ACN addresses one of the ten dimensions of the system, and these dimensions no longer tangle or crosscut each other. We observe that the key design rules we added enable us to decompose the system with finer granularity, and all dimensions only depend on the design rules, but not on each other. The improved decomposition in this case leads to a significantly more efficient modular analysis.

Comparing informally produced DSMs with DSMs derived from precise declarative models reveals errors in the manual versions. Take the oblivious design for example. In the manual version, logging aspects do not depend on the functional specifications. However, these dependences should exist as shown in the derived DSM because the implementation changes are very likely caused by the changes in the specification, and the implementation changes influence the aspects. Although we had been very careful when we constructed this DSM manually, we still missed these dependences due to the transitive relation.

## 7. Related Work

In the realm of constraint networks, researchers have developed numerous ways to decompose and cluster problems. Major decomposition methods include conjunctive decomposition [7], disjunctive decomposition [7], tree clustering [6], etc. These methods make use of the structures of constraint graphs to decompose a CN and compose the results. Our work is different. First, our method is not based on traditional constraint graphs. The edges in our CNF graph do not represent concrete logical relations, and the nodes do not represent variable-value pairs. Second, we use a dominance relation to cut edges in a CNF graph. Dominance relations model hierarchical structures in the domain in which the software is being designed. By contrast, most constraint decomposition methods, such as Choueiry's [5], work on inconsistent variable-value pairs.

There are many bottom-up automatic clustering approaches to automatically discover clusters from source code, such as the work of Hutchens and Basili [9], Schwanke [19], Mancoridis [16]. In contrast to these works, our method does not depend on source code analysis, but rather operates in terms of precise but abstract models of

design architectures. Lattix has produced a tool that derives DSMs from source code (particularly call graph) analysis and that can then reveal discrepancies between actual dependence structures and predicates representing desired structural invariants [11, 18]. MacCormack and Baldwin [14] are pursuing a similar approach. While this work appears to be promising in its own right, the extent to which such source code structure can be taken as a proxy for the kinds of abstract architectures that we model in this paper seems limited and remains unclear.

## 8. Discussion and Conclusion

This paper has contributed a *modular* divide-and-conquer method for efficient analysis of technical and economic properties of declarative models of software design architecture. We evaluated this method using two case studies. The result showed dramatic performance improvement, making it possible to move this modeling technique from the realm of theoretical interest only into the realm of potential engineering utility.

An important question that remains unanswered concerns the difficulty of ACN modeling in a human factors and cognitive sense. At this point we can only report on our own experience. A key point is that modeling with ACNs requires abstraction. Deciding what to model was not always easy. For example, HyperCast includes hundreds of files and tens of thousands lines of code. We modeled it using about 30 variables. We found such modeling difficult at first, but easier as we gained some experience. One clear technical issue is that ACN models themselves remain unstructured. We plan to address this notational issue in our future work. Particularly interesting possibilities involve linking of ACN models to more established architecture description languages and notations for representing product line architectures.

## 9. Acknowledgements

This work was supported in part by the National Science Foundation under grants ITR-0086003 and FCA-0429786. We were inspired to pursue a modular approach to the analysis of ACNs by our earlier work with Joanne Bechta Dugan on modular analysis of dynamic fault tree models.

## References

[1] S. Baase and A. V. Gelder. *Computer Algorithms: Introduction to Design and Analysis (3rd Edition)*. Addison Wesley, 3rd edition, Nov 1999.

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

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

[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*, Long Beach, California, USA, Nov 2005.

[5] B. Y. Choueiry and G. Noubir. A disjunctive decomposition scheme for discrete constraint satisfaction problems using complete no-good sets. In *Knowledge Systems Laboratory*, 1998.

[6] R. Dechter and J. Pearl. Tree clustering for constraint networks. In *Artificial Intelligence* 38:353–366, 1989.

[7] E. Freuder and P. D. Hubbe. A disjunctive decomposition control schema for constraint satisfaction. In *Principles and Practice of Constraint Programming, 1st International Workshop, PPCP'93*, Newport, Rhode Island, 1993.

[8] D. Garlan and M. Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, volume 1, pages 1–40. World Scientific Publishing Company, 1993.

[9] D. Hutchens and R. Basili. System structure analysis: Clustering with data bindings. In *IEEE Transactions on Software Engineering*, 11:749-757., Aug. 1995.

[10] D. Jackson. *Software Abstractions : Logic, Language, and Analysis*. The MIT Press, April 2006.

[11] A commercial product. <http://www.lattix.com/>.

[12] 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.

[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*.

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

[16] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, and E. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *In Proc. 6th Intl. Workshop on Program Comprehension*, 1998.

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

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

[19] R. Schwanke. An intelligent tool for re-engineering software modularity. In *In Proc. 13th Intl. Conf. Software Engineering.*, 1991.

[20] 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.

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