A

A Formal Model for Automated Software Modularity and Evolvability Analysis[1]

YUANFANG CAI, Drexel University
KEVIN SULLIVAN, University of Virginia

Neither the nature of modularity in software design, characterized as a property of the structure of dependencies among design decisions, or its economic value are adequately well understood. One basic problem is that we do not even have a sufficiently clear definition of what it means for one design decision to depend on another. The main contribution of this work is one possible mathematically precise definition of *dependency* based on an *augmented constraint network* model. The model provides an end-to-end account of the connection between modularity and its value in terms of options to make adaptive changes in uncertain and changing design spaces. We demonstrate the validity and theoretical utility of the model, showing that it is consistent with, and provides new insights into, several previously published results in design theory.

Categories and Subject Descriptors: D.2.10 [**Software Engineering**]: Design

General Terms: Design, Evolution, Modularity

Additional Key Words and Phrases: Design Modeling, Design Analysis, Software Economics, Augmented Constraint Network, Design Structure Matrix

ACM Reference Format:
Cai, Y. and Sullivan, K. A Formal Model for Automated Software Modularity and Evolvability Analysis ACM Trans. Softw. Eng. Methodol. V, N, Article A (January YYYY), 29 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Software evolvability, which is achieved in large part through appropriate *modularity in design*, is a non-functional property of great technical and economic importance. Specifying and reasoning about it remains difficult, however, due to the lack of formal models that link design structure, viewed as dependencies among design decisions and the dynamic environments surrounding designs, with its technical and economic implications.

The lack of such models in turn means that we lack software tools for modeling designs and analyzing them for evolvability. Lacking both models and tools, software designers continue to reason informally and intuitively about adaptive capacity. Parnas's

[1]This paper is extended from our previous conference papers [Cai and Sullivan 2005; Sullivan et al. 2001], and some of the prose and figures are from them.

The work of Yuanfang Cai has been supported in part by the National Science Foundation under grants CCF-1065189, CCF-0916891, and DUE-0837665. The work of Kevin Sullivan was supported in part by the National Science Foundation under grants CCF-0613840, and CCF-1052874.
Authors' address: Yuanfang Cai, yfcai@cs.drexel.edu, Computer Science, Drexel University, Philadelphia, PA 19104.
Kevin Sullivan, sullivan@cs.virginia.edu, Computer Science, University of Virginia, Charlottesville, VA 22903.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

information hiding criterion has been influential for decades [Parnas 1972]. Baldwin and Clark's design rule theory [Baldwin and Clark 2000] has shed additional light on the value of design modularity. Sullivan et al. [Sullivan et al. 2001] showed that Baldwin and Clark's model can be extended with environment parameters to account for Parnas's information hiding criterion. However, these accounts remain informal, are unnecessarily hard to understand and hard to apply with rigor and precision.

In this paper, we present a formal model of the fundamental connections among modularity in design structure, the dynamics of the environments that surround designs, the dynamics of design adaptation, and the economic value of adaptive capacity. In more detail, we present a set of inter-linked modeling formalisms. The basic model represents a design space as a first-order theory comprising a set of decisions, possible values for the decisions, constraints on these decisions, the clustering of these decisions into proto-modules, and a dominance relation on decisions. We call this model an *augmented constraint network* (ACN).

Based on ACN modeling, we contribute a precise predicate representing what it means for one decision to depend on another in such a setting. This definition depends on the translation of the ACN into an operational, state-machine form, in which states represent consistent sets of decisions (valid designs), arcs represent changes in individual decisions, and transitions encode minimally disruptive changes in design states. We call this model as a *Design Automaton* (DA). Next we show that such a state-based model can be reduced to a *design structure matrix* (DSM) [Steward 1981; Eppinger 1991; Baldwin and Clark 2000]. We validated this definition by experimental tests: deriving DSMs from ACN models produced by careful reading of source materials, and comparing these new DSMs against ones previously produced by informal means.

Next, we demonstrate that the models are both consistent with, and provide new insights into, long established informal (but nonetheless fundamental) notions of modularity in design. We exploit DSM representations derived from ACN models to link our work to the work of Baldwin and Clark on the economic value of adaptive capacity achieved through modularity in design. Along the way, we develop a formal predicate describing what it means, in the setting of our limited *theory*, for a design modularization to be an *information hiding* modularization in the sense of Parnas. We also demonstrate the feasibility of formalizing Parnas's changeability analysis as a procedure for determining change impact on design decisions.

Finally, these precisely specified modeling formalisms and transformations provide the basis for an automated tool prototype that we call *Simon*[2]. We present the models and the tool together as evidence in support of the claim that it is possible to establish adaptive capacity as a property that can be treated with engineering rigor. To test the proposition that our formalisms and definitions are appropriate we present formalizations of classic case studies in design analysis. In this paper, we represent our work as a starting point, and discuss its limitations in Section 7.

The rest of this paper is organized as follows. Section 2 reviews the background of this work. Section 3 illustrate the framework using a running example. Section 4 presents how previous theories can be formalized within the setting of our framework. Section 5 presents our prototype tool. Section 6 presents case studies. Section 7 discusses the results and limitations of the presented work. Section 8 discusses related work. Section 9 presents our future work and Section 10 concludes.

2. BACKGROUND

In this section we briefly introduce previous work that our model is based upon: Parnas's information hiding criteria [Parnas 1972] and changeability analysis, Baldwin

[2] `http://simon.cs.virginia.edu/`

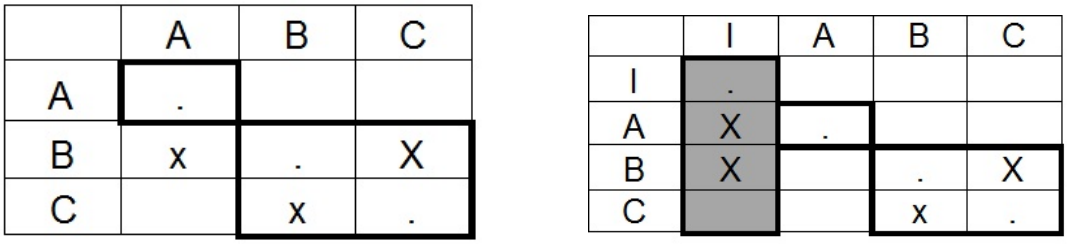
and Clark's design rule theory and design structure matrix modeling [Baldwin and Clark 2000], and how Sullivan et al. used the design rule theory, supported by extended DSM modeling, to precisely capture Parnas's criterion of information hiding [Sullivan et al. 2001].

Parnas's Modularity Theory. In his seminal paper on information hiding [Parnas 1972], Parnas used the design of a *Key Word in Context* (KWIC) program (a program to compute permuted indices) to comparatively analyze two modularization techniques: a traditional design based on the sequence of abstract steps in converting the input to the output, and a new one based on information hiding. The new design uses abstract data type interfaces to decouple key design decisions involving data structure and algorithm choices so that they can be changed without unduly expensive ripple effects. He postulates changes and assesses how well each modularization can accommodate them, measured by the number of modules that would have to be redesigned for each change. He finds that the information-hiding modularization is better, in that given the postulated changes, fewer modules need to be changed. He concludes that designers should use information hiding as a criterion to decompose a design into modules: a module should be characterized by its knowledge of a design decision which it hides from all others, and its interface or definition should be chosen to reveal as little as possible about its inner workings. The information hiding criterion has been influential for decades, but remains intuitive and informal. The informality of the information hiding notion has several consequences, among which is that there is no precisely defined predicate for distinguishing information hiding from non-information hiding design structures, and thus no possibility to automate such an analysis function.

Design Rule (DR) Theory and Design Structure matrix (DSM). Baldwin and Clark's theory [Baldwin and Clark 2000] is based on the idea that modularity adds value in the form of real options. In particular, modularity creates options to make adaptive changes to design decisions within modules at a cost that is kept low by virtue of the decoupling of decisions within modules from decisions within other modules of a system. An option provides the right to make an investment in the future, without a symmetric obligation to make that investment. Because an option can have a positive payoff but need never have a negative one, an option has a positive present value. Baldwin and Clark proposed that a module creates an option to invest in a search for a superior replacement and to replace the currently selected module with the best alternative discovered, or to keep the current one if it is still the best choice. Intuitively, the value of such an option is the value that would be realized by the optimal experiment-and-replace policy.

Baldwin and Clark used *design structure matrices* (DSM) [Eppinger 1991; 1991] as the fundamental model of their option-based modularity theory. DSMs represent and depict, in matrix form, pair-wise dependencies between dimensions of a design space. The columns and rows of a DSM are labeled with design variables modeling design dimensions in which decisions are needed. A mark in a cell indicates that the decision on the row depends on the decision on the column. Design variables for software could represent choices of data structures, algorithms, type signatures, user interface look-and-feel, real time response characteristics, security aspects, power usage, etc. Figure 1 presents a sample design with three variables, A, B, and C. B and C are *interdependent*, resulting in symmetric marks. Algorithm and data structure choices are naturally paired, for example.

A DSM captures two key design activities: the clustering of related design parameters into *proto-modules* and corresponding design tasks, and applying a modular operator, splitting, to decouple design parameters and the corresponding design tasks. A group of interdependent design parameters is clustered into a *proto-module* to show that the decisions are managed collectively as a single design task. In Figure 1, dark



(a) DSM for proto-modular design (b) DSM for a modular design obtained by splitting.

Fig. 1. Design Structure Matrix

lines denote proto-module clusters. A proto-module is essentially a composite design parameter. To be a true module there should be no marks in the rows or columns outside the bounding box of its cluster that connect it to other modules or proto-modules in the system.

Merely clustering a monolithic design into proto-modules does not produce a modular design. In Figure 1 (a), for example, B depends on A, and so the BC proto-module depends on the A proto-module. This dependence can be eliminated, permitting choices for A and B to be made independently, by introducing a new parameter, I, as illustrated in Figure 1 (b). I stands for some kind of interface. An interface allows modules to depend on certain visible decisions while prohibiting dependencies on other, hidden, decisions. In this way, interfaces decouple hidden decisions within one module from hidden decisions in other modules. B no longer depends on A. Instead both take on a dependence on I. A key instance of this operation in software is the introduction of an abstract data type module interface. I would represent an *A-interface* parameter. The B implementation parameter would be constrained to access A only through the interface represented by the value of I. Thus, A could change its implementation freely without affecting B, as long as A's interface did not have to be changed as well.

A design variable, such as I, that decouples otherwise dependent proto-modules, is a *design rule* when this variable is assigned a value and the assigned value is relatively stable, i.e., not subject to changes at high frequency. Design rules impose constraints that other parameters must respect and that they can assume to be stable. Design rules constrain and structure the design space and search process. In Figure 1 (B), we use light grey background in Column I to signify that I is a design rule for this design.

Modularization through the imposition of *design rules* is a key to structuring the design space and search. The purpose of introducing I is to break the dependencies between proto-modules. In a DSM that models a truly modularized system, dependencies only exist within blocks or between blocks and design rules. In Baldwin and Clark's terminology, a *design rule*, such as I, is a (or is part of) a visible module; and a module that depends only on design rules is a *hidden module*.

A hidden module can be adapted or improved without affecting other modules by the application of another modular operator called *substitution*. Baldwin and Clark's theory defines a model for reasoning about the value added to a base system by modularity. This model states that splitting a design into m modules increases its base value S_0 by a fraction obtained by summing the *net option values* (NOV) generated by the m modules. NOV is the expected payoff of exercising a search and substitute option optimally, accounting for both the benefits and costs of exercising options.

The NOV value of a design depends on how it is split into modules. One extreme is to put all decisions into one module. This incurs the highest cost of single experimentation because this module is most complex, which offsets the potential benefits brought

by optimal substitution. On the other extreme, each decision can have its own module. Although each module thus has least complexity and hence lowest experimental cost, the low technical potential of each module and the cost of ripple effects caused by the interdependencies among these modules offset the potential benefits obtained from searching and substitution.

Extended Design Rule theory and DSM model. Sullivan et al. [Sullivan et al. 2001] showed that DSMs as used by Baldwin and Clark and in earlier work do not appear to model the environment in which a design is embedded, making it impossible to model the forces that drove design changes, and thus do not provide enough information to justify estimates of the environment-dependent technical potential parameters of the NOV model. Sullivan et al. thus extended the DSM modeling framework to model what we call *environment parameters* (EP). *Design parameters* (DPs) are endogenous: under the control of the designer. Even design rules can be changed, albeit possibly at great cost. However, the designer does not control EPs. They are exogenous.

Sullivan et al. used the extended DSM model to capture Parnas's two modularizations of KWIC, and applied Baldwin and Clark's substitution NOV model to compute quantitative values of the two modularizations [Sullivan et al. 2001], using parameter values derived from information in the DSMs combined with the judgments of a designer. They showed that the extended DSM model provides a clear visual representation of genuine information hiding. Information hiding is indicated when the sub-block of an DSM where the EPs intersect with the DRs is blank, signaling that the design rules are invariant (not sensitive, or coupled, to) with respect to changes in the environment. In this case, only the decisions hidden within modules have to change when EPs change, not the design rules. They also showed that the information hiding design generates much higher net options value than the sequential design.

Remaining Challenges. Notwithstanding the demonstrated strengths of DSMs, they remain inadequate in the following ways. First, they are ambiguous, in the sense that the meaning of dependence between design decisions is not well defined. This ambiguity, in turn makes it difficult, time-consuming, and error-prone to decide exactly what dependencies to mark in a DSM [Cai and Sullivan 2005]. Second, they are too incomplete, i.e., abstract, to account for the contents of informal design theories such as those of Parnas and Baldwin and Clark. In particular, although a DSM does represent design dimensions and dependencies between them, it abstracts from both the choices available in each dimension and from the constraints on these choices that induce the exhibited dependencies. One consequence of this incompleteness is that a DSM does not explicitly model the multiple ways in which, in general, a given change in design can be accommodated. Indeed, in the presence of multiple possible compensations, the very meaning of a dependence mark in DSMs becomes unclear: does a mark mean *must change* or *is subject to change in some scenario* or *could be changed but does not have to be*?

3. ANALYTICAL MODELING FRAMEWORK

This section introduces the following concepts: the *Augmented Constraint Network* (ACN), *Design Automata* (DA), and *Pair-Wise Dependency Relation* (PWDR), using an example of Irwin et al. [Irwin et al. 1997][3]. Irwin et al. and subsequently Kiczales et al. [Kiczales et al. 1997] considered the case of a software component for storing and manipulating matrices. The key idea is that the best choice for an underlying data structure depends on the needs of the client, to manipulate either spare or dense matrices. The high-valued points in the design space occur when the selected data struc-

[3]We have formalized all the concepts introduced in this section in Zed, which can be found at: `http://simon.cs.virginia.edu/Spec`

ture is consistent with the user's needs: e.g., the requirement is for dense matrices, and contiguous arrays of memory cells are used to store the data, or the requirement is for sparse matrices, and some kind of linked representation is selected. Our work considers the consequences of changes in decisions. If the overall system were in a good, high-density state but the user requirements change to a low-density state, how should the overall system be adapted? In this section, we discuss our approach to representing design spaces of this kind, as well as transitions between high-value points in such spaces, driven by changes in externally-driven variable values. We use the example to provide a full picture of the core models: *augmented constraint network* (ACN), *design automaton* (DA), and *pair-wise dependence relation* (PWDR), representing decision-making phenomena from different perspectives.

3.1. Augmented Constraint Network

The core computation model of an ACN is a *constraint network* (CN) [Mackworth 1977]. A constraint network consists of a set of *variables*, V, with their *domains*, D, and the constraints, C, among these variables: $CN = \langle V, D, C \rangle$. To model a conceptual design, each *variable* models a design or relevant environmental dimension. The *domain* of a *variable* comprises a set of *values*, each representing a possible decision or an environmental condition.

We augment ACN with two additional data structures to formally account for two design activities that are not readily representable in a constraint network: the *dominance* relation among decisions and the aggregation of design decisions into *candidate modules*, each of which represents one possible way a design can be modularized.

We observe that the essence of Baldwin and Clark's notion of design rule is the *dominance relation* (DR) among design decisions: some design decisions dominate other subordinating decisions. For example, the decisions to add a feature dominate the other subordinating decisions to realize the feature. $(x, y) \in DR$ indicates that, due to policy or lack of control, changes in x cannot be compensated for by changes in y (even if changes in y can be accommodated by changes in x).

Module is another essential concept in software design that a constraint network does not lend itself to modeling. The net option value of a design depends on how the design is split into modules. Given the same design, clustering it in different ways will generate different net option values, because important parameters in Baldwin and Clark's NOV formula, such as the complexity and dependents of each module will be different. We model the multiple *modularization candidates* using a *Cluster Set* (CS) that consists of a set of *clustering*. Each clustering expresses *a priori* aggregation of subsets of variables into candidate modules. The same design can have different clustering methods, reflecting different stakeholders' views of the design.

These augmented data structures have played important roles in Baldwin and Clark's DSM-based modularity analysis, and we found that formalizing these notions and combining them with a constraint network provide additional analysis power. We call a constraint network augmented with a *dominance relation* (DR) and a *cluster set* (CS) an *augmented constraint network* (ACN), formally: $ACN = \langle CN, DR, CS \rangle$.

In Figure 2, we show how to use a constraint network to model the design of the example program used in Irwin et al.'s and Kiczales et al.'s work [Irwin et al. 1997; Kiczales et al. 1997]. The design is under the context of a *client* who needs to represent and manipulate a matrix that can be either sparse or dense. We abstractly model the design as having two dimensions, the data structure and the associated algorithm. The scalar variables, *ds* and *alg* in lines 2 and 3, model these dimensions; the *client* variable in line 1 represents the client demand, the density. Their domains follow within the parentheses. We use *other* as a value in many domains to model unelaborated other

```
1: scalar client: (dense,sparse);
2: scalar ds: (list_ds,array_ds,other);
3: scalar alg: (array_alg,list_alg,other);
4: (ds = array_ds) => (client = dense);
5: (ds = list_ds) => (client = sparse);
6: (alg = array_alg) => (ds = array_ds);
7: (alg = list_alg) => (ds = list_ds);
```

Fig. 2. The Constraint Network for the Small Example [Irwin et al. 1997]

possibilities. In Figure 2, for example, Line 2 models that the choices for the data structure dimension include array, list, and other unelaborated choices.

A set of logical constraints in a constraint network models the interdependence relation among design variables and environmental conditions. Line 4 in Figure 2 states that the choice of an array representation is valid only if the client needs dense matrices. Logically, the decision on the left of the implication assumes the decision on the right. This might seem counterintuitive, but there could be other data structure choices that are also consistent with density, and we do not want to model an overly constrained design in which *array* is the only choice.

The *binding* of a *value* to a *variable* models a design decision or an environmental condition. An *assignment* is a set of bindings, modeling a set of given decisions or environment conditions. For example, $\{client = dense,\ ds = array\_ds\}$ is an assignment. A *valuation* is a *complete assignment* involving all the variables in the ACN. $\{client = dense,\ ds = array\_ds,\ alg = array\_alg\}$ is a valuation.

A *valid design* is a *solution* to the constraint network, that is, a valuation that satisfies all the constraints defined in the ACN. All the valid designs constitute a *design space* as modeled by a given ACN [4].

In the small example, assuming that the client's need dominates and that the design decisions must adapt accordingly, the *dominance* relation thus includes the following two pairs: *(ds, client)* and *(alg, client)*.

There are multiple ways a design can be clustered into modules. In the small example, we can view all the three variables as one module, each variable as a module, or cluster any two of the variables as one module. To model the fact that whether the client's need is out of the designer's control, we put *client* into an *environment* cluster and put the other two variables into a *design* cluster.

3.2. Operational Design Space Evolution Model

A logic-based design description is not sufficient for reasoning about design evolvability and economic properties. From an ACN model, we derive an operational, state-machine-based evolution model called the *design automaton* (DA) that represents the *change dynamics* within a *design space*, based on the propagation of changes through the constraint network of an ACN.

A DA can be derived from the constraint network and the dominance relation of an ACN. The state set of a DA is the *design space* and consists of all the solutions of an ACN; the transitions of a DA model the design variation constrained by the ACN. Figure 3 shows the partial DA of the small example. The designs are numbered and constitute the state set of the DA. The circles in Figure 3 model all the valid designs within the design space.

[4] In general, there are many possible dimensions for a given set of requirements outside of the space modeled by an ACN. Baldwin and Clark use the term *design space* to refer to the larger space of all possibilities. In this sense, an ACN is an explicit representation of a subspace of interest.

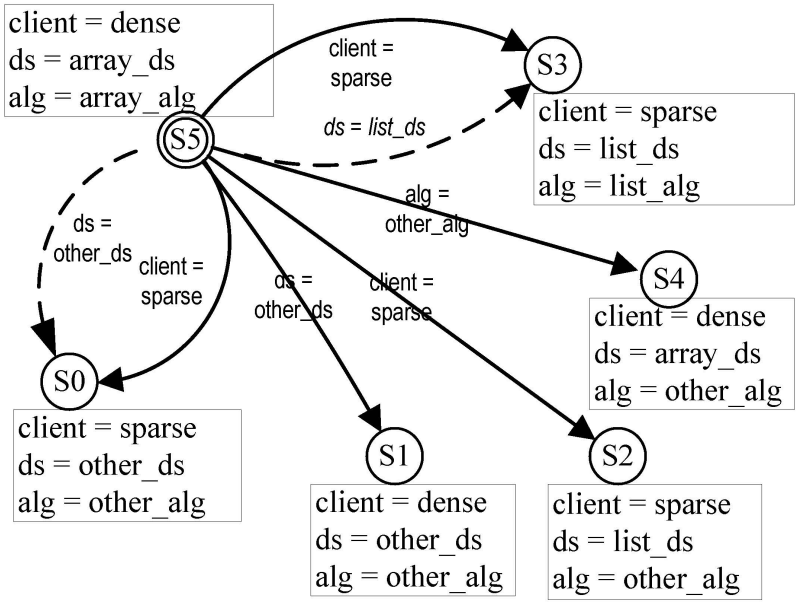


Fig. 3. Partial Design Automaton

Changing the value of one design decision can produce a complete assignment that violates one or more constraints. For example, if we start with the design $S5$ in Figure 3, $\{$*(client = dense), (ds = array_ds), (alg = array_alg)*$\}$, and change the data structure, *ds*, to *list_ds*, the resulting state violates a constraint, producing an invalid design state. In order to restore the design to a consistent design state, then, in general, the values of some subset of other variables will have to change. In this case, both *ds* and *alg* have to be changed. Figure 3 depicts part of the DA where all changes are originated from design $S5$, illustrating three key properties of a DA:

(1) Each transition in a DA is *minimal*. That is, each destination state differs only minimally from the previous state, in the sense that no constituent change could be undone while still preserving consistency. In Figure 3, starting with $S5$, if *ds* is changed to *other_ds*, then there are at least two designs that can accommodate this change: *S0* and *S1*. Changing *alg* to *other_alg* in both $S0$ and $S1$ is indispensable, but changing *client* to *sparse* in design $S0$ is not. We consider the transition from $S5$ to $S1$, labeled with $\{$*ds = other_ds*$\}$, as *minimal*, while the dotted arrow transition from $S5$ to $S0$ with the same label is invalid. As a result, each transition in a DA models a *minimal design perturbation*.

(2) A DA is *nondeterministic*. In general, there are multiple ways to accommodate a change. In Figure 3, starting from state $S5$, changing the client preference to *sparse* makes the design inconsistent. Making a set of *minimal* changes to other variables to restore consistency leads to states S0, S2, or S3.

(3) No transition in a DA may violate the *dominance* relation. If $(x, y) \in$ *dominance*, then among all the possible ways to restore consistency in the face of a change to x, those involving y are excluded. For the small example, because $(ds, client) \in$ *dominance*, the transition starting from $S5$, triggered by changing *ds* to *list_ds*, and leading to the client change in S3 (the dotted arrow labeled *ds = list_ds*) is precluded.

In summary, a DA captures all of the possible ways in which any change to any decision in any state of a design can be compensated for by changes to minimal subsets of other decisions.

3.3. Pair-wise Dependence Relation

Pair-wise Dependence Relations (PWDRs) underlie many influential design representations. In box-and-arrow style representations, such as architectural description languages (ADLs), the unified modeling language (UML), and call graphs, the arrows model different kinds of pair-wise dependence relations among boxes, such as function calls, inheritance, and system I/O. A *Design Structure Matrix* also represents a PWDR on design decisions.

Based on the DA model, we contribute a precise definition of what it means for one variable to depend on another, enabling the automated derivation of PWDRs from DAs. Intuitively, for some consistent design state s in a DA, if there is some change to a variable, x, such that the value of another variable, y, is changed in some minimally perturbed destination state s' of the DA, we say that *y depends on x*. We define the *coupling structure* of a design ACN as the *pair-wise dependence relation* (PWDR) over all of its variables.

In the given example, if the original design is *S5* and the envisioned change in client is *(client = sparse)*, there are three new designs accommodating this change in its DA: *S0, S2, S3*. Comparing the original design $S0$ with any of these new designs, we observe that both *ds* and *alg* are involved in the minimal perturbations caused by the change to *client*. That is, both *ds* and *alg* depend on *client*. Similar analysis concludes that *ds* and *alg* depend on each other. As a result, the matrix PWDR is the following set: $\{(client, ds), (client, alg), (ds, alg), (alg, ds)\}$.

The three core models, ACN, DA, and PWDR connect to (but are not limited to) a number of well-known evolvability and economic analyses, such as Parnas's changeability analysis and information hiding criterion, as well as Baldwin and Clark's net option value analysis, providing the foundation to automate these analysis techniques using tools.

4. FORMULATING MODULARIZATION THEORIES USING ACN MODELS

In this section, we show how the models we presented in Section 3 formally account for previous modularity theories.

4.1. Parnas's Theory

We now show how Parnas's information hiding criterion and changeability analysis can be formalized in our ACN-based models.

Information Hiding Modularity. Sullivan et al. [Sullivan et al. 2001] present characterization of the nature of Parnas's information hiding modularity as invariance of design rules with respect to changes in environment variables in a Baldwin-and-Clark-style DSM. We can formalize these concepts in our ACN-based formal model: after partitioning the variable set of an ACN, V, into three subsets: *Environmental Variables* (*ENVR*), *Design Rules* (*DR*), and *Hidden Variables* (*HV*). If the ACN models an information hiding modularized design, its derived *pair-wise dependency relation* (PWDR) should not have any pair with the first element in *ENVR*, and the second in a *DR*.

Identifying which dimensions belong to environment, design rule, or hidden variable scope is not always easy. Tool-supported ACN modeling allows the user to cluster variables in different ways according to their own understanding about how the system should be clustered into modules, which parts are out of designer's control, and which parts should remain stable.

Changeability analysis. Given a current design, what are all the ways to compensate for a sequence of given decision changes? Parnas's changeability analysis to find the ripple effects of a change can be inferred from the answer to this more general question by comparing the feasible new designs with the original design. The answer to this question has the potential, for example, to find the most cost-effective way to accommodate a change.

We formalize the question as a function *impact* that maps an original design and a sequence of changes to a set of evolution paths, each comprising of a sequence of designs accommodating the changes. The start design is the first state in each of the evolution paths. Each transition step consumes a change. The last designs of these paths are the new designs that the original one could reach.

4.2. Baldwin and Clark's Theory

We now show that how certain key concepts in Baldwin and Clark's modularity theory, especially concerning the substitution of one hidden module for another, can be formalized based on our ACN modeling, how to automatically derive *design structure matrix* (DSM) from an ACN model, and how to exploit DSM modeling to link logical models with option-based analysis.

Formalizing the key concepts of the design rule theory. We formally accounted for Baldwin and Clark's concepts of *design dimension*, *design decision*, *design space*, *design dependence*, and *design rule* as follows:

— Design dimensions: *Variables* of an ACN;
— Design decisions: *Values* of a variable;
— Design spaces: the solutions of a constraint network, that is, the state set of a DA [5].
— Design dependencies: the PWDR model that formally defines what it means by saying one decision depends on another.
— Design rules (in part): the dominance relation models a property of design rules [6].

Automatic DSM derivation. The derivation of a DSM from a PWDR becomes straightforward. We define a DSM as a PWDR and a clustering: $DSM = \langle PWDR, clustering \rangle$. A *PWDR* is used to populate the matrix: if $(x,y) \in PWDR$ then there is a mark in row y and column x. A clustering is used to order the columns and rows of the matrix. Different DSMs can be derived from a given PWDR using different clusterings. According to this definition, a DSM with precise semantics can be automatically derived. Important properties of a design decision, such as which other variables depend on it, can be automatically calculated.

Linking logical models with option theory. Baldwin and Clark's theory defines a model for reasoning about the value added to a base system by modularity. This model states that splitting a design into m modules increases its base value S_0 by a fraction obtained by summing the *net option values* (NOV) (NOV_i) of each module. NOV is the expected payoff of exercising a search and substitute option optimally, accounting for both the benefits and cost of exercising options. The value of a software with m modules is calculated as:

$$V = S_0 + NOV_1 + NOV_i + ... + NOV_m, \text{ where}$$
$$NOV_i = max_{k_i}\{\sigma_i n_i^{1/2} Q(k_i) - C_i(n_i)k_i - Z_i\}$$

[5] Technically speaking, the design space for a problem is the entire set of possible designs that address that problem. A DSM or ACN generally models only a sub-space. When we say that we formalize the notion of design space, we mean that we have formalized an approach to modeling such sub-spaces.
[6] According to Baldwin and Clark, design rules dominate other design decisions, decouple otherwise dependent decisions, and remain stable. This paper does not intend to formalize the stable property.

For module i, $\sigma_i n_i^{1/2} Q(k_i)$ is the expected benefit to be gained by accepting the best positive-valued candidate generated by k_i independent experiments. $C_i(n_i)k_i$ is the cost to run k_i experiments as a function C_i of the module complexity n_i. $Z_i = \Sigma_j c n_j$, where module j depends on module i, is the cost of changing the modules that depend on module i. The max picks the experiment that maximizes the gain for module i. We summarize all the parameters needed in the NOV model and explain how to derive some of these parameters from the ACN model.

The first parameter is the number of modules m, which can be calculated as the number of clusters in a given clustering.

Second, the complexity of the whole system and each module are needed. Baldwin and Clark used the number of variables within a DSM and each block to model the complexity. In an ACN model, the complexity of the whole system can be calculated as the number of variables of the ACN. The *Complexity* of a module can be measured as the size of the module as a proportion of the overall system. This is not the only way complexity can be measured. We use this measurement to make it consistent with Baldwin and Clark's complexity measurement based on DSMs because ACN variables can be directly mapped to the columns and rows of a DSM. We made this choice also because we need to make the NOV analysis consistent and comparable with our previous analysis based on informal DSM models [Sullivan et al. 2001].

Third, the user needs to estimate the cost of each experiment, which is beyond the scope of ACN, and can be viewed as a user input.

Fourth, the *visibility cost* measures the cost incurred by dependencies between modules. From our PWDR model, we can automatically calculate the dependency relation among modules. In Baldwin and Clark's design rule book, it is not clear whether the visibility cost should also include indirect ripple dependencies. In our PWDR model, all the ripple dependencies are already taken into account by the underlying constraint network. As a result, the concept of *visibility* is formalized. On the other hand, the cost of modifying each dependent module still relies upon user estimation and input.

Finally, the most important parameter for NOV analysis is the *technical potential*, σ, of each module, which is the expected variance on the rate of return on an investment in producing variants of a module implementation. On the assumption that the prevailing implementation of a module is adequate, the expected variance in the results of independent experiments is proportional to changes in requirements that drive the evolution of the module's specification. This is another parameter that has to reply upon user estimation and input.

In summary, from an ACN model, we can automatically derive a DSM and calculate all the structure-related parameters needed by the NOV model. Some of the NOV parameters inevitably require user estimation and input, which can be supported by an ACN modeling and analysis tool, linking logical-based design abstraction with economic analysis.

5. SIMON: THE TOOL

In this section, we introduce our prototype tool, called Simon [7], that allows the user to build ACN models and to conduct a number of analyses. Simon supports ACN modeling through interactive graphical user interfaces (GUIs), and automates design impact analysis, design structure matrix derivation, and net option value calculation. Figure 4 shows the relations among the core models of our framework and the automated analysis techniques supported by Simon.

[7]Our tool is named after Herbert A. Simon, a pioneer of decision making theories and artificial intelligence. A new version of Simon is available to the research community at the Simon web site: `http://simon.cs.virginia.edu`.

5.1. Formal Design Modeling

Using the GUIs of Simon, the user can input the three elements of an ACN, a constraint network, a dominance relation, and a cluster set, through a tab control, as shown in Figure 5, 6, and 7. The user can save the ACN project as a set of files.

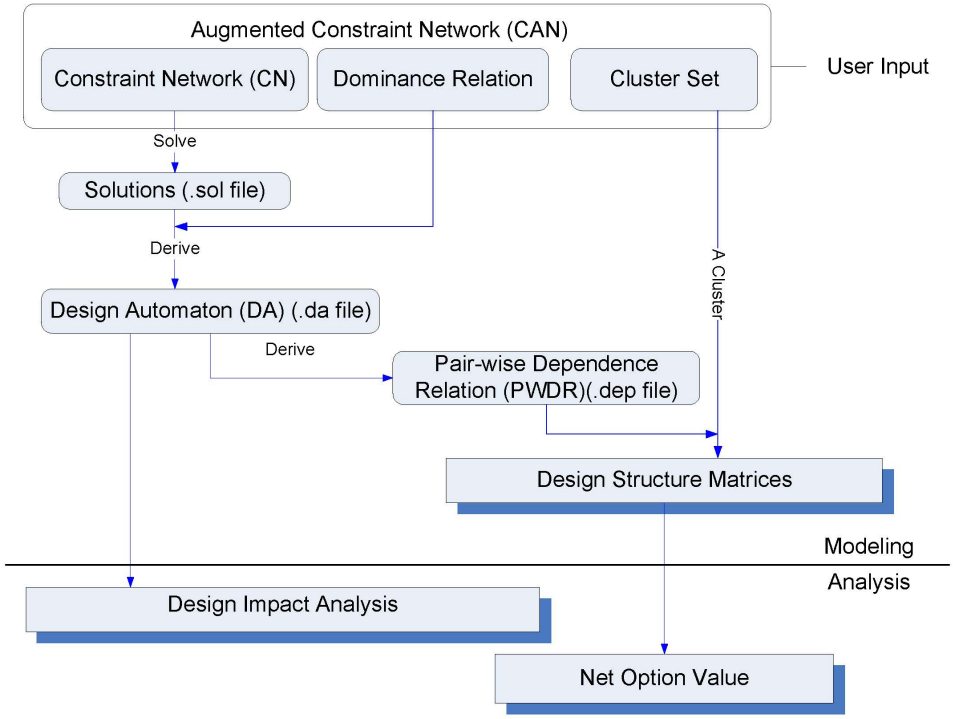


Fig. 4. Core Models and Analysis in Simon

Figure 5 shows the constraint network tab page. The left list box shows all the design variables, and the right list box shows all the constraints. The domain of a selected scalar variable is shown in the lower left box of the tab page. The user can add new variables or constraints through pop-up windows. The underlying parser checks the syntax and semantic off the new variables and constraints. Valid variables and constraints are added to the underlying ACN data structure and displayed in the GUI. Figure 6 shows the dominance relation tab page, in which the user can construct the dominance relation through a grid control. Checking a cell dictates that the variable on the row can not influence the variable on the column. Figure 7 shows the cluster set tab page, in which the user can create, delete, or edit a cluster by moving variables around and aggregating variables into clusters. Newly added clusters are shown in the upper left *cluster set* box of the form, which displays existing clusterings. The *selected cluster* boxes display the selected cluster. Selecting a cluster reorders the variables in display accordingly.

Given an ACN, Simon first solves the constraint network using Alloy [Jackson 2002] internally. We wrote a small helper program in Java to invoke the SAT solver through Alloy APIs, and translate Alloy outputs into a text file (a .sol file) in the format Simon requires. After that, Simon takes the solutions and the dominance relation as input,

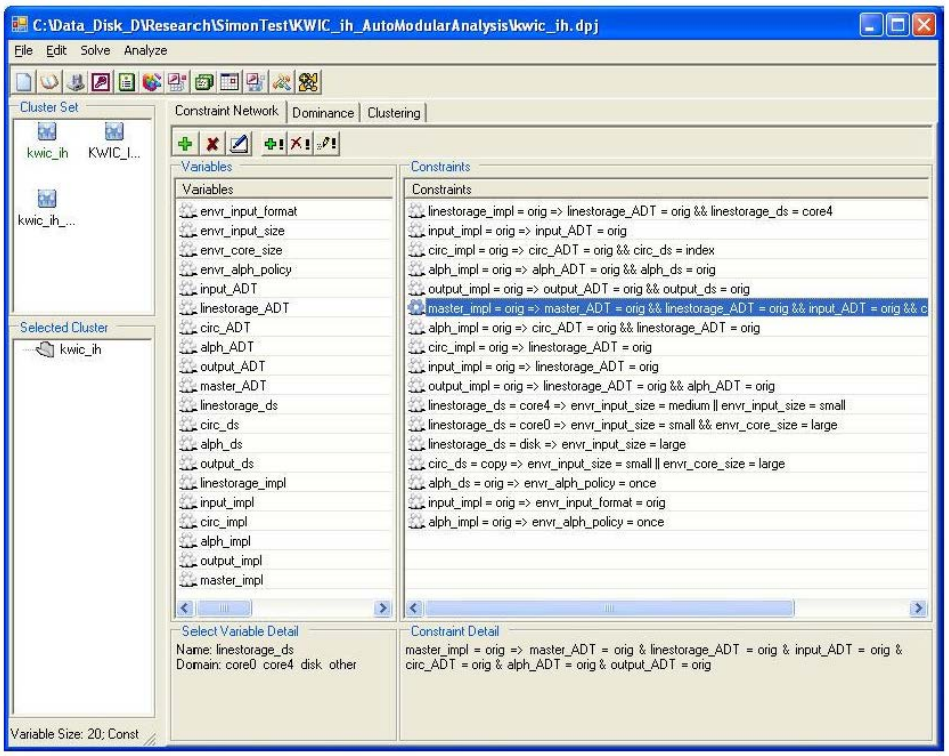


Fig. 5. Simon: Constraint Network Construction

generates a design automaton and a pair-wise dependence relation, and stores them into separate files (a .da file and a .dep file). For performance reasons, we wrote a DA and PWDR generation program in C. Constraint solving, solution enumeration and DA generation are all time consuming. Simon uses a divide-and-conquer algorithm for solution enumeration and DA generation [Cai 2006].

5.2. Automated Analysis

Given a solved ACN and the computed DA and PWDR relation, the user can analyze design impacts using the GUIs (Figure 8, 9, and 10), derive design structure matrices (Figure 11), and compute net option values (Figure 12).

Change Impact Analysis. The inputs of design impact analysis include an original design and a sequence of changes, and the output includes a set of evolution paths. Figure 8 shows the Simon GUI in which the user can specify a design by selecting a value for each variable. Clicking the *Verify* button tests whether the specified design is valid. Given a valid design, the user can specify a change by selecting another value for a changing variable using the GUI shown in Figure 9. All the changed variables are shown in the lower list view as a sequence. The user can then click the *Analyze* menu to analyze design impact and get the output in a GUI as shown in Figure 10. The upper block shows two evolution paths. Clicking the corresponding radio button shows the selected design in the evolution path in the lower box. The middle list view shows the differences between the original design and the selected design in the evolution path.

Design Structure Matrix Derivation. Clicking the *Design Structure Matrix* menu generates a DSM, as shown in Figure 11. A DSM is derived from a PWDR and a selected

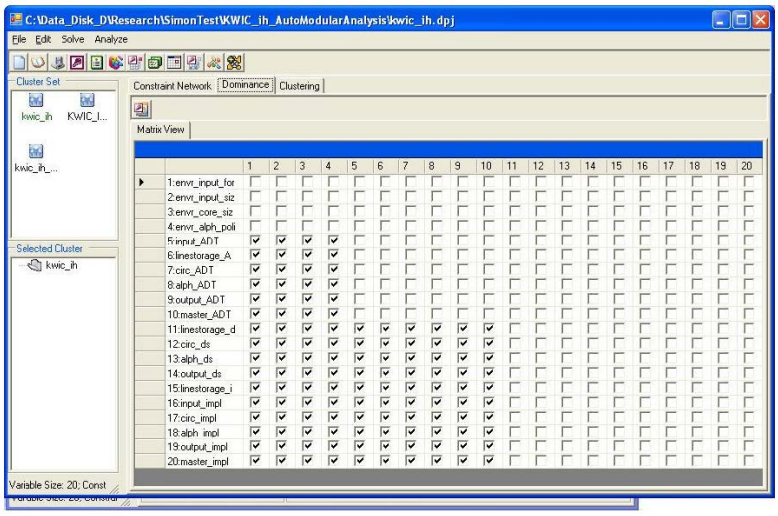


Fig. 6. Simon: Dominance Relation Construction

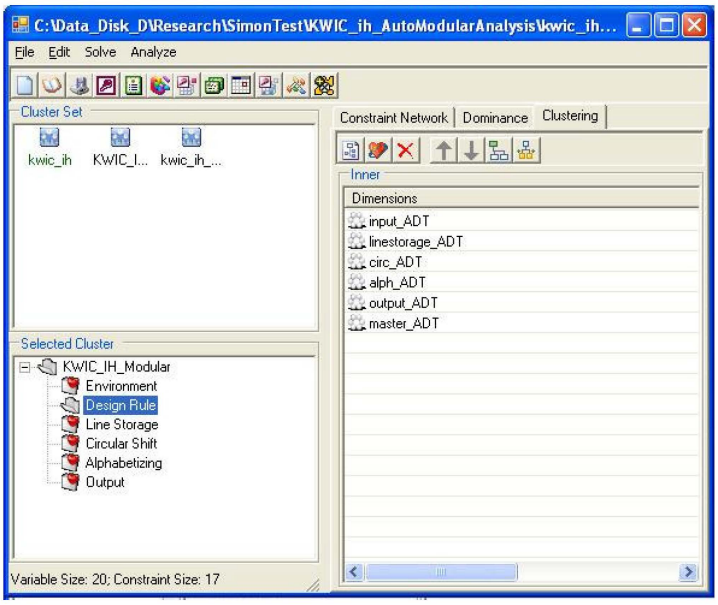


Fig. 7. Simon: Cluster Set Construction

clustering from the clustering set. Selecting a different clustering method reorders the DSM accordingly.

Net Option Value Computation. Clicking the *NOV* menu in the DSM GUI reveals the net option value calculation GUI, as shown in Figure 12. Of all the parameters that are needed for NOV calculation, the following can be automatically derived: the number

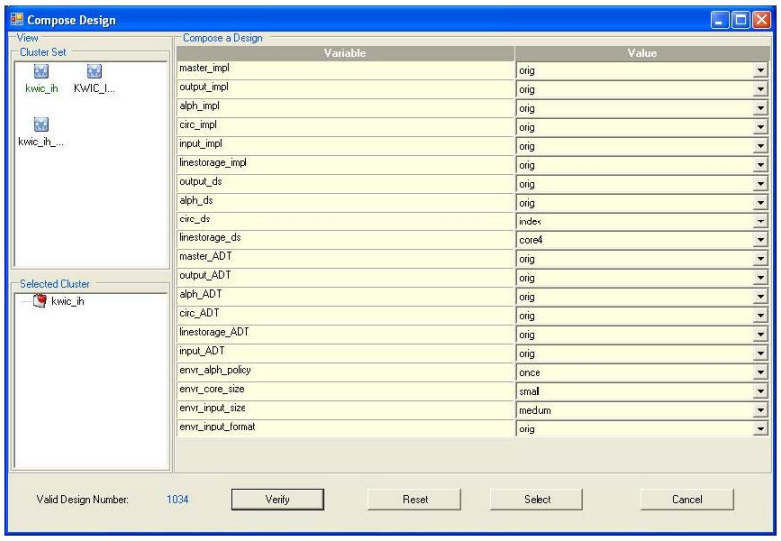


Fig. 8. Design Impact Analysis: Select an Original Design

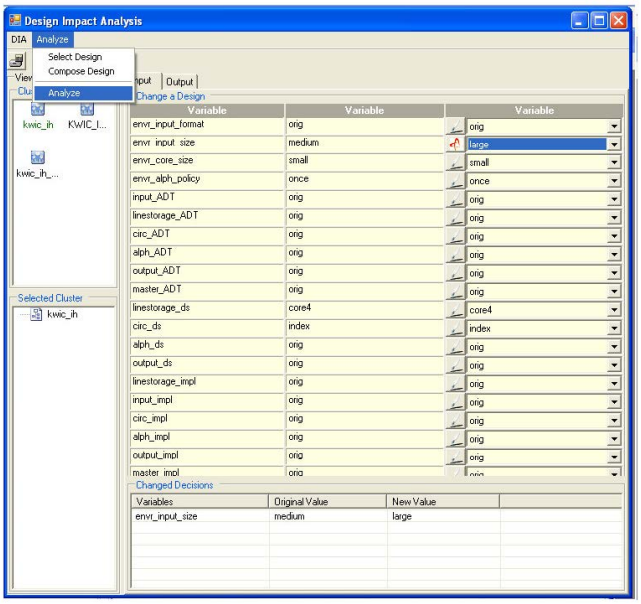


Fig. 9. Design Impact Analysis: Specify a Change

of modules can be derived from the cluster used to order the DSM; the *complexity* of a module can be approximated by the number of variables within the module divided by total number of variables of the ACN; and the dependents of a given module, used to calculate the cost of experimenting a module, can be derived from the PWDR rela-

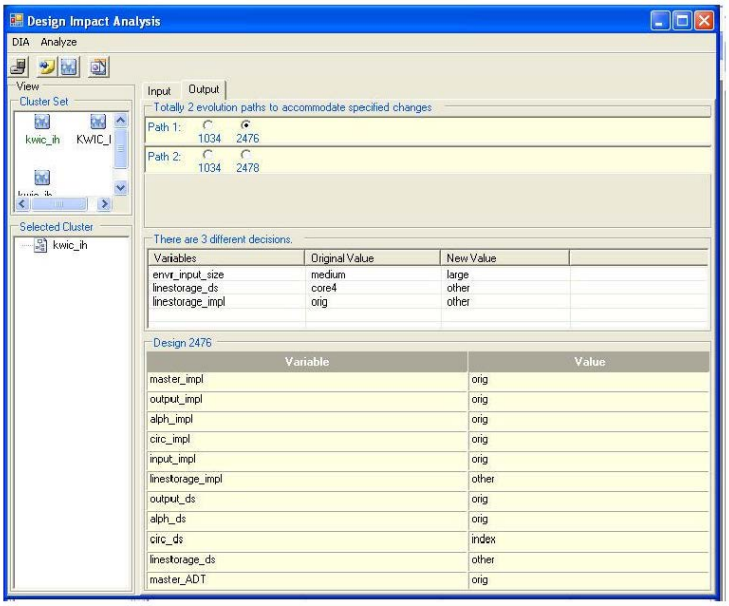


Fig. 10. Design Impact Analysis Output: Evolution Paths

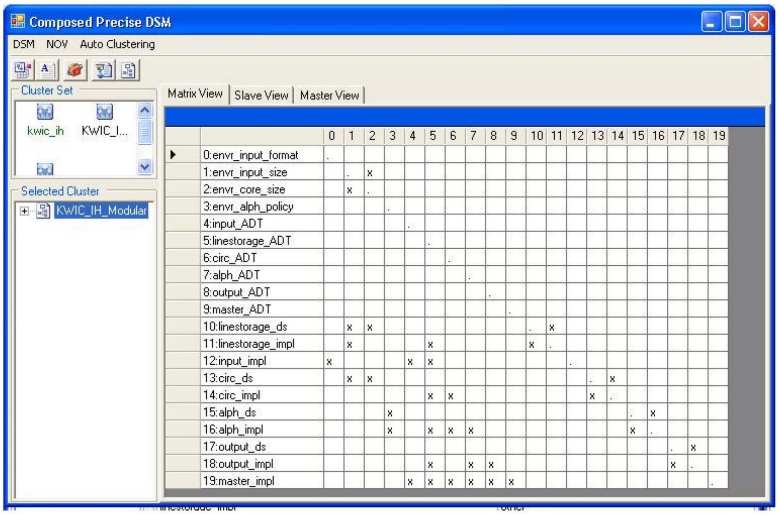


Fig. 11. Design Structure Matrix Derivation

tion. The user can input additional NOV related parameters, including the estimated technical potential and the estimated cost of conducting experiments on each module.

This GUI displays module parameters according to the selected clustering. The user can experiment with a new modularization method by creating a new cluster using the GUI shown in Figure 7, and then computing the corresponding NOV. The upper right

block summarizes all the parameters of all the modules. The lower right grid shows the automatically computed NOV values for each module. The final system NOV is shown at the bottom.

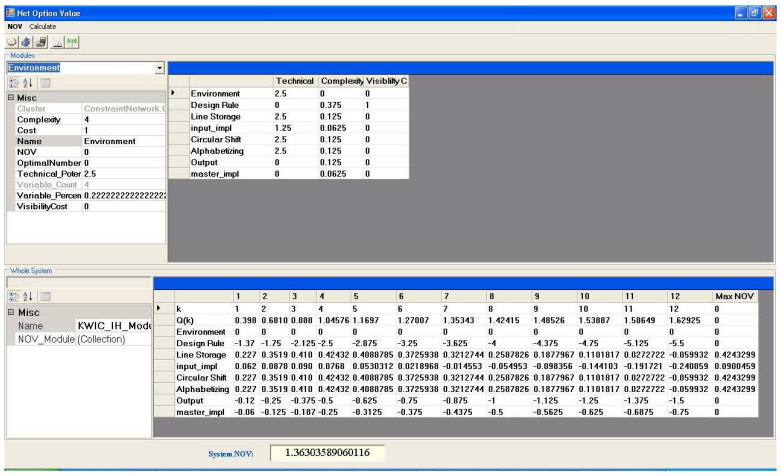


Fig. 12. Net Option Value Calculation

In summary, this function connects structural properties automatically derived from logical design models with the economic value of the design, allowing the user to take uncertainty into consideration by estimating technical potential of each module according to their domain knowledge.

6. CASE STUDY: PARNAS'S KWIC

In this section, we evaluate the theoretical utility of the ACN model using Parnas's (Key Word in Context) (KWIC) [Parnas 1972] system. We show that the ACN model can formalize Parnas's information hiding criterion using Baldwin and Clark's design rule theory and design structure matrix modeling, Simon can automate Parnas's changeability analysis, and the computed results verify or correct previously informal analysis results. We have applied ACN modeling to a number of larger scale case studies for different purposes [Cai and Sullivan 2006; Cai et al. 2007; Huynh et al. 2008; Wong et al. 2009; Wong and Cai 2009; Wong et al. 2011]. In this paper, we use KWIC, a small but well-known and well-established benchmark, to illustrate the formalization of previously informal design principles.

A KWIC index system accepts an ordered set of lines; each line is an ordered set of words, and each word is an ordered set of characters. Any line may be *circularly shifted* by repeatedly removing the first word and appending it at the end of the line [Parnas 1972]. The KWIC system outputs a listing of all circular shifts of all lines in alphabetical order.

Parnas comparatively and informally analyzed two designs for KWIC. In the first sequential design (SQ), modules correspond to steps in the sequential transformation

of inputs to outputs. In the second, information hiding (IH) design, modules decouple design decisions deemed complex or likely to change. The IH design uses abstract data type interfaces to decouple key design decisions involving data structure and algorithm choices so that they can be changed without unduly expensive ripple effects. Parnas presents a comparative analysis of the changeability of these two designs. He postulates changes and assesses how well each design can accommodate them, measured by the number of modules that have to be redesigned for each change. He finds that the information-hiding modularization is better in that fewer number of modules needs to be revisited.

We modeled and studied these two designs using traditional DSMs [Sullivan et al. 2001]. The published quantitative analysis and manually-constructed DSMs provide suitable basis for explication and result comparison. Using Simon, we evaluate our framework through the following steps:

(1) Design Modeling: we develop an ACN model for each design of KWIC in order to answer several questions. Do these ACNs have the expressive capacity to capture key design issues and constraints in an abstract and informative way? Is this modeling method adequate to model environmental impacts?
(2) DSM derivation: we derive DSMs for each design to answer the following questions: are the automatically generated DSMs consistent with the manually-constructed version in the precious work [Sullivan et al. 2001]? Do they similarly reveal the key observations made there? If there is inconsistency, which one is more faithful in revealing important dependency properties such as ripple effects? Where does the inconsistency come from?
(3) Design impact analysis (DIA): we apply DIA analysis to quantify Parnas's qualitative comparative analysis of the changeability of the two KWIC designs. We first formalize the five possible changes he postulates as decision problems, then use Simon to compute the multiple ways these changes can be accommodated. After that, we compare the differences between the original design and new designs to see whether the quantitative results are consistent with Parnas's qualitative analysis.

6.1. Design Modeling

We model the two KWIC designs as two augmented constraint networks, each comprising a constraint network, a dominance relation, and a cluster set.

Constraint Network. For the SQ design, Parnas describes five modules: Input, Circular Shift, Alphabetizing, Output, and Master Control. He views each interface as providing two parts: an exported data structure, and a function signature to be invoked by Master Control. Given choices for these variables, programmers produce function implementations. We modeled the choices of function signature, data structure, and implementation as *design variables*.

As shown in Figure 13, variables ending with "*_sig*" model function signatures. The choices of implementation are modeled by the variables ending with "*_impl*". The choices of data structures are modeled by variables ending with "*_ds*". Parnas assumes original designs in each case and analyzes the impact of changes. We use value *orig* in most of the domains to model Parnas's original choices. We extend each design variable domain with another value, *other*, to model the choice that is different than the original one but not yet elaborated. For example, the *input_fun_sig* has domain {*orig, other*}.

We represent the relationships among these design decisions as logical constraints. In the SQ design, function implementations make assumptions about both function signatures and relative data structures. For example, the circular shift function im-

```
DesignSpace kwic_seq{
1   envr_input_format:{orig,other};
2   envr_input_size:{small,medium,large};
3   envr_core_size:{small,large};
4   envr_alph_policy:{once,partial,search};
5   input_sig:{orig,other};
6   circ_sig:{orig,other};
7   alph_sig:{orig,other};
8   output_sig:{orig,other};
9   master_sig:{orig,other};
10  input_ds:{other,core4,disk,core0};
11  circ_ds:{index,copy,other};
12  alph_ds:{orig,other};
13  output_ds:{orig,other};
14  input_impl:{orig,other};
15  circ_impl:{orig,other};
16  alph_impl:{orig,other};
17  output_impl:{orig,other};
18  master_impl:{orig,other};
19  input_impl = orig => input_sig = orig
      && input_ds = core4;
20  circ_impl = orig => circ_sig = orig
      && circ_ds = index;
21  alph_impl = orig => alph_sig = orig
    && alph_ds = orig && circ_ds = index;
22  output_impl = orig => output_sig = orig
      && output_ds = orig;
23  alph_ds = orig => circ_ds = index;
24  alph_impl = orig => input_ds = core4;
25  circ_impl = orig => input_ds = core4;
26  circ_ds = index => input_ds = core4;
27  circ_ds = copy => input_ds = core4;
28  output_impl = orig => input_ds = core4;
29  output_impl = orig => alph_ds = orig;
30  alph_ds = orig => input_ds = core4;
31  input_ds = core4 => envr_input_size = medium
      || envr_input_size = small;
32  input_ds = core0 => envr_input_size = small
      && envr_core_size = large;
33  input_ds = disk => envr_input_size = large;
34  circ_ds = copy => envr_input_size = small
      || envr_core_size = large;
35  input_impl = orig => envr_input_format = orig;
36  alph_impl = orig => envr_alph_policy = once;
37  master_impl = orig => input_sig = orig
    && circ_sig = orig && alph_sig = orig
    && output_sig = orig  && master_sig = orig;
38  alph_ds = orig => envr_alph_policy = once;
}
```

Fig. 13. SQ Design Constraint Network

plementation has to know the circular shift function signature and how the circular shift data is arranged in the memory. According to Parnas, this function operates data through index in the original design. We model this choice as a value of *circ_ds*, *index*. Line 20 in Figure 13 models the constraint. To implement this function, it also has to know the data structure of the *Input* module. In the current design, the characters are packed four to a word, which is modeled as a value, *core4*, of variables *input_ds*. The constraint is modeled in Figure 13 line 25.

Figure 14 shows the constraint network for the IH design. A new module, *Line Storage*, is present. Its data structure variable *linestorage_ds* replaces the *input_ds* in the SQ design. The input module no longer has separate data structure. In the IH design, each module is also equipped with an abstract data type interface, modeled by variables ending with "*_ADT*". Module implementations and data structures are modeled in the same way. According to Parnas's paper, a module only knows the ADTs of other modules. For example, *circ_impl* now assumes *linestorage_ADT*, as shown in Figure 14 line 28, but not the data structure any more.

Parnas presents a comparative analysis of the *changeability* of the two designs based on their ability to accommodate the changes in the following environmental conditions: the input format, input size, core size and alphabetizing policy. We model these environmental conditions as the environment variables. In both designs, the variables starting with "*envr_*" are environment variables.

The environmental conditions also have domains modeling different possibilities. For example, the domain of *envr_core_size* is {*small, large*}, and the domain of *envr_alph_policy* is {*once,partial,search*}. Data structure and implementation usually

```
DesignSpace kwic_ih{
1   envr_input_format:{orig,other};
2   envr_input_size:{small,medium,large};
3   envr_core_size:{small,large};
4   envr_alph_policy:{once,partial,search};
5   input_ADT:{orig,other};
6   linestorage_ADT:{orig,other};
7   circ_ADT:{orig,other};
8   alph_ADT:{orig,other};
9   output_ADT:{orig,other};
10  master_ADT:{orig,other};
11  linestorage_ds:{core0,core4,disk,other};
12  circ_ds:{copy,index,other};
13  alph_ds:{orig,other};
14  output_ds:{orig,other};
15  linestorage_impl:{orig,other};
16  input_impl:{orig,other};
17  circ_impl:{orig,other};
18  alph_impl:{orig,other};
19  output_impl:{orig,other};
20  master_impl:{orig,other};
21  linestorage_impl = orig => linestorage_ADT = orig
      && linestorage_ds = core4;
22  input_impl = orig => input_ADT = orig;
23  circ_impl = orig => circ_ADT = orig
      && circ_ds = index;
24  alph_impl = orig => alph_ADT = orig
      && alph_ds = orig;
25  output_impl = orig => output_ADT = orig
      && output_ds = orig;
26  master_impl = orig => master_ADT = orig
      && linestorage_ADT = orig
      && input_ADT = orig && circ_ADT = orig
      && alph_ADT = orig
      && output_ADT = orig;
27  alph_impl = orig => circ_ADT = orig;
28  circ_impl = orig => linestorage_ADT = orig;
29  input_impl = orig => linestorage_ADT = orig;
30  output_impl = orig =>alph_ADT = orig;
31  linestorage_ds = core4 => envr_input_size = medium
      || envr_input_size = small;
32  linestorage_ds = core0 => envr_input_size = small
      && envr_core_size = large;
33  linestorage_ds = disk => envr_input_size = large;
34  circ_ds = copy => envr_input_size = small
      || envr_core_size = large;
35  alph_ds = orig => envr_alph_policy = once;
36  input_impl = orig => envr_input_format = orig;
37  alph_impl = orig => envr_alph_policy = once;
    }
```

Fig. 14. IH Design Constraint Network

make assumptions about the relative environmental condition. Line 34 and 35 in Figure 13 are two examples.

Dominance Relation. We now identify the design rules in both designs according to Parnas's description. In the SQ design, Parnas noted, "*All of the interfaces between the four modules must be specified before work could begin...*" These are the choices of function signatures and data structures that dominate other design variables, the design rules of the SQ design. The SQ *dominance* relation thus includes pairs like: *(input_fun_impl, input_fun_sig)*, *(input_fun_impl, input_ds)*, etc. Similarly, in the IH case, the choices of ADT interface definitions dominate other decisions, serving as design rules, and pairs like *(linestorage_ds, linestorage_ADT)* are thus set in the IH *dominance* relation.

In both designs, we assume that the environmental conditions are out of the control of the designers. Accordingly, *(linestorage_ds, envr_input_size), (linestorage_ds, envr_core_size)*, etc., are included in the *dominance* relations of each design.

Cluster Set. There are multiple ways to cluster a design. Figure 15 shows the Simon clustering GUI supporting different views of the same KWIC design. For purposes such as task assignment, we want to group all variables involved in a particular function into a single module. For example, we could group the *envr_alph_policy*, *alph_ADT*, *alph_ds* and *alph_impl* into a module, as shown in Figure 15 (b).

In our earlier work [Sullivan et al. 2001], we observed that for a design to be truly an information hiding modularization, the design rules should be invariant under changes in the environment variables. To evaluate these two designs against this criterion, we want to cluster the environment variables, design rules and subordinate vari-

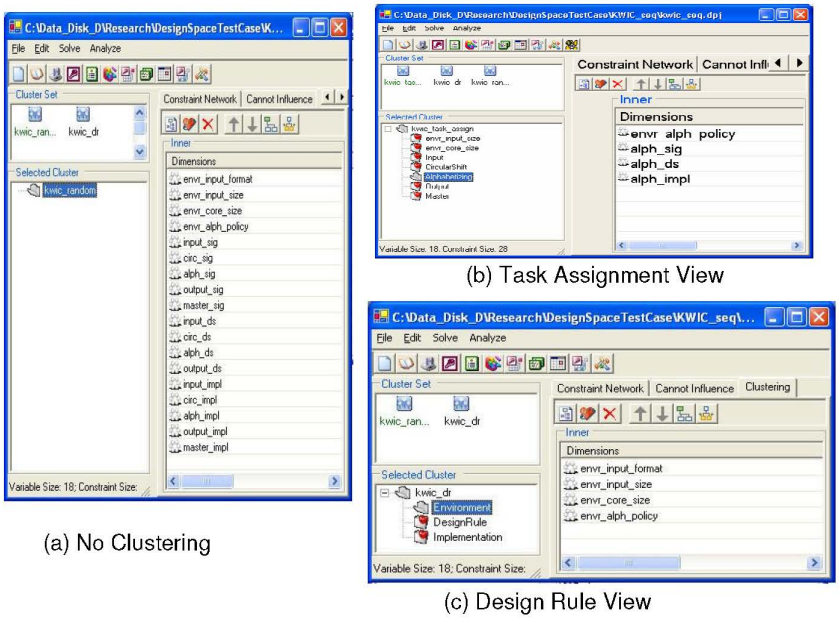


Fig. 15. Simon clustering GUI for IH Design

ables respectively into proto-modules. In this case, we group the four environmental variables, *envr_alph_policy*, *envr_input_size*, *envr_core_size*, and *envr_input_format*, into an *environment* module, as shown in Figure 15 (c).

So far, we obtained positive answers for the questions proposed at the beginning of this section: the modeling approach is expressive enough to capture key design issues and constraints, to model environmental impacts and to represent unknown regions abstractly and informatively.

6.2. DSM Derivation

After the DA and PWDR are generated, the user is able to view the generated DSMs and analyze design change impacts. We compare the DSMs that Simon generates from our ACN models with the manually-constructed DSMs we previously presented [Sullivan et al. 2001]. To ease the comparison, we copied and pasted the DSM generated from Simon into a spreadsheet and marked the differences from the manually-constructed DSMs, as shown in Figure 16 (SQ) and Figure 17 (IH).

In these figures, the light grey blocks along the diagonal are modules. The first block (variables 1–4) contains environment variables. The next block (variables 5-13) contains design rule variables. The next five light grey blocks model hidden modules. The dark grey area below the design rule block models how design rules influence the hidden modules. The dark grey area to the right of the design rule block is blank, showing the dominating roles of these design rules. That is, the hidden decisions do not influence design rules. All the cells with black background represent the discrepancies between derived and manually-constructed DSMs. A blank black cell means that there

was an erroneous mark in the manually-constructed version. A black cell with an ”x” in it means that the dependence was missed in the manually-constructed version.

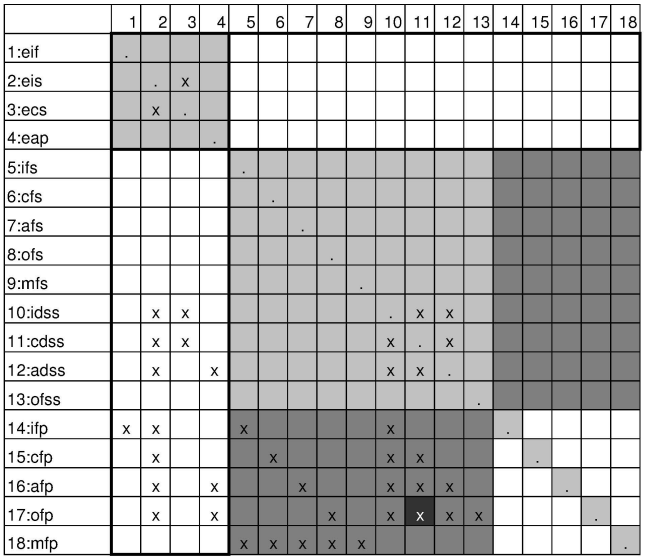


Fig. 16. Sequential Design DSM

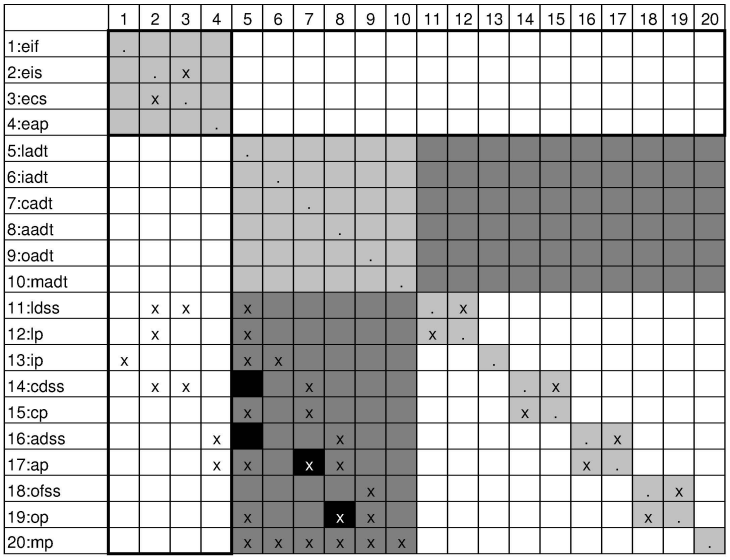


Fig. 17. Information Hiding Design DSM

By comparison, we are able to answer the validation questions for DSM derivation. First of all, our computed DSMs were largely consistent with the earlier results, validating the modeling and analysis concept. They reveal exactly the same key observa-

tions: the design rules should be invariant with respect to changes in the environment and that such changes should be accommodated by changes to hidden (subordinate) design variables.

There are differences, however, which we now address. First, the computed DSMs reveal subtle errors in our manually produced DSMs, supporting our intuition that logic modeling and automated analysis are more reliable than manual modeling and analysis. In our computed information hiding (IH) DSM, cells (17, 7) and (19, 8) revealed dependences missing from our manually-constructed DSM. It also lacks several dependences that should not have been present in the manually-constructed version. An extra variable, *input_ds*, which is redundant with *linestorage_ds*, was removed. Finally, the environment variables *core_size* and *input_size* are also now shown as dependent, in that a change in one can be compensated for by a change to the other.

The second class of differences between Simon's output and our manual calculation consists of important ripple effects in the computed DSMs that are not shown in the manually-constructed version. For example, our manually-constructed sequential design (SQ) DSM had no dependence between *output_fun_impl* and *circ_ds*. The derived DSM revealed this dependence owing to two constraints in its ACN model:

$$output\_fun\_impl = orig \Rightarrow alph\_ds = orig$$
$$alph\_ds = orig \Rightarrow circ\_ds = index$$

Parnas's paper confirms the presence of this dependence and the correctness of the formal model and derived DSM. Even in such a small and well-studied example, the manually-constructed DSM is error-prone. Automated tool support is critical for correct modeling and analysis of complex design constraint networks.

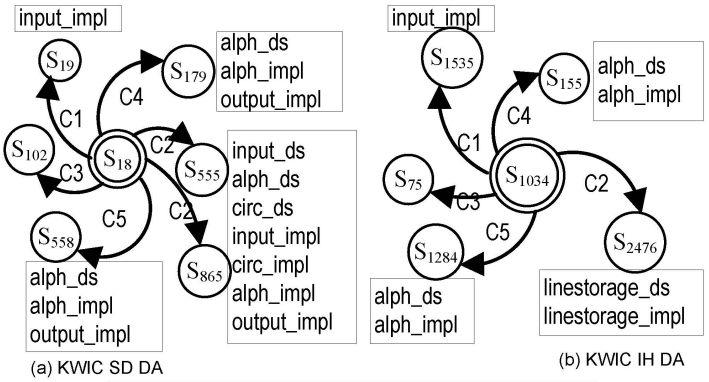
6.3. Design Impact Analysis

Parnas comparatively analyzed these two designs by considering four possible changes. We model these changes using our framework as follows:

(1) *The input format changes*. It implies that there could be other input format choice other than the current one. Accordingly, we model the domain of *envr_input_format* as {*orig, new*}. In the original design, *envr_input_format* = *orig*. The change is modeled as *envr_input_format* = *new*.
(2) *The input size becomes so large that not all lines can be put in core.* We model this change as *envr_input_size* = *large*.
(3) *The input size gets so small that a word could be unpacked.* We model this change as *envr_input_size* = *small*. These two changes imply that the input size of the original design is medium, and the domain of *envr_input_size* is {*small, medium, large*}.
(4) *The alphabetizing policy is changed to* partial *or* search, modeled by *envr_alph_policy* = *partial* and *envr_alph_policy*=*search*. Originally, *envr_alph_policy* = *once*.

Parnas's informal comparative analysis can be formulated as follows: given an original design, and given changes in the environment (input size, core size, etc.), what are the feasible new designs that accommodate the given changes, and how many modules have to change to get to these new design states?

We organize all the changes and their impacts on both designs computed by Simon into Figure 18. The numbers in the circles represent the design states of the DAs. The double circles are the start states. It shows part of the SQ and IH DAs with states 18 and 1034 as the respective start states. Transitions are labeled with changes shown in the table below.



	Change	*SD*	*IH*
C1	envr_input_format = new	1	1
C2	envr_input_size = large	7	2
C3	envr_input_size = small	0	0
C4	envr_alph_policy = partial	3	2
C5	envr_alph_policy = search	3	2

Fig. 18. Partial Non-deterministic Finite Automaton for SQ and IH design

The tables associated with the end states show what other variables are changed in the destination states. For example, in the SQ DA, changing the input size to large (the transition labeled C2) leads state S_{18} to state S_{555} or S_{865}. In both of them, 7 other variables are changed to compensate for the driving change.

The numbers in the last two columns of the lower table summarize the number of other variables that are affected by the changes in each design. The results confirm in a fully formal way that the IH design space involves fewer redesign requirements. For example, when the input size gets large, in SQ design, 7 dimensions has to be touched, while for IH design, only 2 variables need to be revisited. So far, we have quantitatively confirmed Parnas's qualitative analysis.

7. DISCUSSION

In this section, we discuss the theoretical utilities of our formalizations, the scalability of the approach, as well as other related issues.

Theoretical Utility. We developed the formal definitions to clarify what it means for design decisions to be dependent, and what information is needed to represent modular design structure in a declarative form but in a way that is also adequate to derivation of DSM and related models. The formalization served to (a) support the development of definitions, (b) as a specification for the automation of Simon, and (c) thereby to support automated experimental validation.

In a nutshell, we view this work as a first step toward a formal theory connecting design structure, viewed in decision-theoretic terms, to models of the dynamics of design evolution (the DA), and to the value of adaptive capacity in designs (Net Options Value). While Baldwin and Clark did formalize the financial aspects of their model, they were not able to be rigorously precise about design structure because they lacked a set of definitions such as the ones we developed and validated, and which we present in this paper. Our tool demonstrates that these definitions of ours, combined with the formal financial economic theory of Baldwin and Clark enable for the first time the

development of fully formal models linking structure to option-theoretical techniques for characterizing the economics of flexibility in design.

The specification provides precise, formal, declarative representations of *structured design spaces*. Using an ACN to generate a DSM, the user will avoid making such mistakes as marking a wrong cell or missing an indirect dependency. However, it is unavoidable to make other kinds of mistakes when manually constructing ACN models. One of the common problems we have experienced is to overlook the existence of direct dependencies. Consequently, constraints can be missed from the resulting ACN model. In the case study reported in this paper, all the direct dependency are derived from the existing manually constructed DSMs and Parnas's descriptions. Our recent work has formalized and automated the transformation from prevailing design models, such as the UML component diagram and class diagram, into ACN models [Wong and Cai 2009; Avritzer et al. 2010][8].

Scalability. As with many formal analysis techniques, such as model checking [Clarke et al. 2000], the difficulty of constraint satisfaction limits the size of models that can be analyzed in practice. Our DA model requires an explicit representation of the entire space of satisfying solutions, but the number of the solutions increases exponentially with the number of variables involved. For example, the ACN that models Parnas's KWIC information hiding design has 20 variables and 34907 solutions. It took hours for the first version of Simon to get the analysis results.

To address this problem, we created a method to decompose a large ACN model into a number of smaller sub-ACNs, using design rules that are formalized as the *dominance relation* of the ACN, solve each sub-ACN individually, and integrate the analysis results [Cai and Sullivan 2006]. The integrated results are equal to the results obtained by analyzing the full large ACN model. This approach splits the whole KWIC information hiding ACN into 6 sub-ACNs, having 6, 6, 4, 5, 7, and 5 variables respectively. Simon now invokes multiple SAT solvers and DA processors concurrently to deal with these much smaller models, and integrate the results in the order of seconds [Cai and Sullivan 2006].

The decomposition methods enabled us to study systems with larger sizes. We first studied a web application developed and studied by Lopes et al. [Lopes and Bajracharya 2005] (WineryLocator). In their paper, Lopes et al. use Baldwin and Clark's modeling and analysis technique to quantitatively compare different designs. We represented these designs using ACNs according to their design descriptions, generated DSMs, and compared with their manually constructed DSMs. We found ambiguities and problematic issues in their published manually-constructed models and analysis [Cai and Sullivan 2006]. We also studied a peer-to-peer networking system, HyperCast [Liebeherr and Beam 1999; Liebeherr et al. 2002], developed by researchers in the University of Virginia and studied by Sullivan et al. [Sullivan et al. 2005]. Similar to the WineryLocator paper, the authors of the HyperCast paper compared different designs using manually-constructed DSMs. Remodeling these designs into our framework and analyzing them automatically reveals important issues in the manually-constructed models. Recently we have used the framework to quantitatively compare aspect-oriented and object-oriented design of design patterns [Cai et al. 2007]. Using a design DSM generated by an ACN as the model, and a DSM reverse-engineered from the source code as a sample, we have created a genetic algorithm to automatically check the conformance between design and implementation [Huynh et al. 2008]. Using this technique, we have found a number of implementation issues of HyperCast.

Uncertainty. The technical potential of each module is an input parameter to the Simon tool modeling uncertainties within the system. In essence, the formal structural

[8]All these ACN related projects can be found at `http://www.cs.drexel.edu/~yfcai/Projects.html`

model is annotated with technical potential on a per module basis. Simon then computes the NOV of a modular design by first reducing an ACN to a DSM, associating the sigma (technical potential) estimates with each module of the DSM, and then running the resulting data through Baldwin and Clark's NOV formulae.

Design Space Modeling. What we are proposing is an approach to explicit modeling and analysis of design spaces, whereas current development practice tends to be more focused on finding and developing a single point in a design space. (One can argue that in practice designers consider just enough of a design space to find a viable design, and there is certainly truth in this position. What is generally uncommon is any kind of more systematic and explicit modeling and analysis of design spaces. Advances in theoretical and eventually practical work lie in focusing more clearly on design spaces and on dynamical processes of evolution and adaptation in such spaces.)

Dependence Relation and Use Relation. The notion of *dependence* formalized by our ACN model is related to but different from the *use* relation Parnas proposed [Parnas 1979]. The *use* relation can be viewed as one kind of assumption relations between the interfaces of modules. The PWDR formalized in our work covers a broader scope than that of the *use* relation, for example, including the dependency between design dimensions and environmental conditions.

8. RELATED WORK

Our work is related to software architecture research [Garlan and Shaw 1993; Abowd et al. 1995; Taylor et al. 2009]. Most such work is committed to an ontology of components and connectors. Logical variables and constraints are more general and expressive. Our models do capture a notion of architecture in the sense of design decisions (especially design rules) on which much depends. Stafford and Wolf [Stafford and Wolf 2001] studied architectural dependence analysis for architecture definition languages. Our work, in contrast, is not confined to architectural level, and is formal and automatable. In addition, ADLs support such analyses as the compatibilty between components and connector, while our framework aims to support evolvability and modularity analysis.

Batory [Batory and O'Malley 1992] uses formal models of software design spaces for systems that vary in component implementations. Their work aims to support system generation and reuse. Jackson [Jackson 2002] used Alloy for object modeling with the goal of automatically proving properties of given models. Abowd et al. [Abowd et al. 1995] used Z to formalize architectural styles in order to prove mainly behavioral properties of systems in these styles. Other researchers are exploring the use of constraint networks in design space search [Lane 1990; Sinnema et al. 2004] and optimization for complex embedded system design [Mohanty et al. 2002]. The goal is to find good designs under constraints. Our aim and contribution, by contrast, is a theory of coupling in logical design space models.

Traditional impact analysis research focuses on change issues at source code level, as summarized in [Arnold and Bohner 1996]. We have provided a precise notion of impact analysis at *design level*. The commonality between our work and traditional model checking work is that both use SAT solver and model design spaces. Our work differs in that we model design decisions and environmental conditions, not the states of components. The constraints we model are mainly the assumption relations among design decisions and environmental conditions. The dominance relation in an ACN are not part of traditional model checking. Finally, our purpose is to conduct change impact analysis and provide precise definition of pair-wise dependencies, rather than discovering optimal solutions.

The notion of *dominance relation* we introduced is somewhat related to the work of Borning et al. on constraint hierarchies [Borning et al. 1991]. Different from their

work, we model dominance relation as a data structure independent of but complementary to the constraint network, formalizing the notion of *design rules*.

9. FUTURE WORK

We implemented the specification and validated the definitions using our tool. It might be valuable to test the specification directly by encoding it in a language such as Alloy, and by subjecting it to formal verification, which is our future work.

Massive monolithic constraint networks are clearly not the right way to go. We are working on theoretical constructs and supporting software tools to support the refinement, abstraction, incremental development, and evolution of modular components of such constraint networks.

There are now commercial DSM-based software modeling tools, such as Lattix [Sangal et al. 2005], that derive dependence information from static analysis. The resulting dependencies are generally syntactic and static semantic dependencies between source code objects, not dependencies between more abstract design decisions. It would be interesting to look at what would amount to reverse engineering from source code DSMs to models at the level of more abstract design decisions. We will explore this possibility in the future.

In our current formalization, the dominance relation is not transitive. But in some cases it seems that one would want to interpret it transitively. In other cases perhaps not. One possible future work would be to provide the user the option to specify dominance relation either transitively or non-transitively.

10. CONCLUSION

To develop an engineering discipline for software requires stronger scientific foundations [Shaw 1990]. Such a science demands theories, models, and experiments [Leveson 1994]. The main contribution of this paper is a first, albeit quite abstract and simplified, formal theory of connections between design structure in a decision-theoretic setting, the dynamics of design evolution, and the economic value of modularity in design. By way of early validation of this theory, we showed, first, that it is adequate to capture the essential content of several foundational but previously informal characterizations of design phenomena; and, second, that it supports precise specification of automated analysis techniques by which one can derive results of known utility (such as DSMs) from formal, abstract, representations of design spaces.

We have not claimed that our theory or tools are adequate yet for detailed, whole-system modeling or analysis of industrial software in all of its vast complexity. Indeed, there are obvious limitations, not withstanding the adequacy of the theory to the tests to which we have subjected it. First, the design variables that we have modeled to date are not only finite-state but encode decisions of only a few bits each. Real design decisions often have large, possibly continuous, domains, with more complex constraints (e.g., arithmetic). Analysis of dependence structures in such settings is far more complex, to the point of being intractable or uncomputable in the general case. Third, our models abstract to small numbers of decisions variables. Fourth, the models in the theory reported in this paper are static in terms of the decisions variables and constraints in question. Real designers confront evolving sets of variables and constraints. Fifth, we have modeled only highly abstract design spaces. These are some of the limitations of this work with respect to the full complexity of real industrial design.

In short, the work we report here is theoretical in nature. Without formalization of concepts (such as information hiding), using logic and mathematics, it is difficult to claim credibly that one has a real theory; and without theory, it is difficult to claim credibly that one has a genuine science. Intractable theories about issues that are of importance in principle do sometimes lead to results of practical significance, as

for example in model checking. Theory can also deepen our insights in basic issues, e.g., what iS the nature of dependence among software design decisions? Theory can also help us to understand practical limits: e.g., how hard will it be to compute pairwise dependence structures as a function of the kinds of constraints that are used in modeling? To what extent in principle can we reason about dependences among decisions in complex designs?

What we have presented in this paper is a formal theory that we have shown is adequate to account for some essential characteristics of several conceptual models that are widely accepted by researchers as having validity with respect to industrial practice: those of Parnas, and Baldwin and Clark. Ultimately, we do wish for theories to have practical consequences. Some of our recent work suggests that it is worth continuing to explore in this dimension. We used an extended version of our theory [Cai and Sullivan 2006], for example, to model and produce DSMs for larger systems [Wong et al. 2009; Sethi et al. 2009; Wong et al. 2009; Wong and Cai 2009; Wong et al. 2011]. We have also shown [Huynh et al. 2008] the possibility to use our theory for work of potential practical value in architectural compliance checking based on the matching of design-level DSMs derived from ACN models with code DSMs derived by static analysis. We remain guardedly optimistic that attention to the theoretical foundations of software design does have the potential for significant payoffs over time.

ACKNOWLEDGMENTS

We would like to thank Derek Rayside, Christian Estler, and Daniel Jackson for their detailed comments and suggestions. We also thank Carliss Baldwin of the Harvard Business School for her collaborations with us around issues of modularity and evolution in complex system designs.

REFERENCES

ABOWD, G. D., ALLEN, R., AND GARLAN, D. 1995. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology 4,* 4, 319–64.

ARNOLD, R. AND BOHNER, S. 1996. *Software Change Impact Analysis* First Ed. Wiley-IEEE Computer Society Pr.

AVRITZER, A., PAULISH, D. J., CAI, Y., AND SETHI, K. 2010. Coordination implications of software architecture in global software development projects. *Journal of Systems and Software 83,* 10, 1881–1895.

BALDWIN, C. Y. AND CLARK, K. B. 2000. *Design Rules, Vol. 1: The Power of Modularity*. The MIT Press.

BATORY, D. AND O'MALLEY, S. 1992. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology 1,* 4, 355–398.

BORNING, A., MAHER, M., MARTINDALE, A., AND WILSON, M. 1991. Constraint hierarchies and logic programming. In *Proceedings 6th Intl. Conference on Logic Programming, Lisbon, Portugal, 19–23 June 1989*, G. Levi and M. Martelli, Eds. The MIT Press, Cambridge, MA, 149–164.

CAI, Y. 2006. Modularity in design: Formal modeling and automated analysis. Ph.D. thesis, University of Virginia.

CAI, Y., HUYNH, S., AND XIE, T. 2007. A framework and tool supports for testing modularity of software design. In *22nd IEEE/ACM International Conference on Automated Software Engineering*. 441–444.

CAI, Y. AND SULLIVAN, K. 2005. Simon: A tool for logical design space modeling and analysis. In *20th IEEE/ACM International Conference on Automated Software Engineering*. Long Beach, California, USA.

CAI, Y. AND SULLIVAN, K. 2006. Modularity analysis of logical design models. In *21th IEEE/ACM International Conference on Automated Software Engineering*. Tokyo, JAPAN.

CLARKE, E. M., GRUMBERG, O., AND PELED, D. A. 2000. *Model Checking*. The MIT Press.

EPPINGER, S. D. 1991. Model-based approaches to managing concurrent engineering. *Journal of Engineering Design 2,* 4, 283–290.

GARLAN, D. AND SHAW, M. 1993. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, V. Ambriola and G. Tortora, Eds. Vol. 1. World Scientific Publishing Company, 1–40. Large-scale architecture patterns: pipes and filters, layering, black-board systems.

HUYNH, S., CAI, Y., SONG, Y., AND SULLIVAN, K. 2008. Automatic modularity conformance checking. In *Proceedings of the 30th International Conference on Software Engineering*. 411–420.

IRWIN, J., LOINGTIER, J.-M., GILBERT, J. R., KICZALES, G., LAMPING, J., MENDHEKAR, A., AND SHPEISMAN, T. 1997. Aspect-oriented programming of sparse matrix code. In *ISCOPE*. 249–256.

JACKSON, D. 2002. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol. 11,* 2, 256–290.

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C. V., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*. Springer-Verlag, 220–42.

LANE, T. G. 1990. Studying software architecture through design spaces and rules. Tech. Rep. CMU/SEI-90-TR-18, CMU.

LEVESON, N. G. 1994. High-pressure steam engines and computer software. *IEEE Computer 27,* 10, 65–73.

LIEBEHERR, J. AND BEAM, T. K. 1999. Hypercast: A protocol for maintaining multicast group members in a logical hypercube topology. In *Networked Group Communication*. 72–89.

LIEBEHERR, J., NAHAS, M., AND SI, W. 2002. Application-layer multicasting with delaunay triangulation overlays. *IEEE Journal on Selected Areas in Communications 20,* 8.

LOPES, C. V. AND BAJRACHARYA, S. K. 2005. An analysis of modularity in aspect oriented design. In *Proceedings of the International Conference on Aspect-Oriented Software Development*. ACM Press, New York, NY, USA, 15–26.

MACKWORTH, A. 1977. Consistency in networks of relations. In *Artificial Intelligence, 8*. 99–118.

MOHANTY, S., PRASANNA, V. K., NEEMA, S., AND DAVIS, J. 2002. Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. *Proceedings of the joint conference on Languages, compilers and tools for embedded systems: software and compilers for embedded systems 37,* 7, 18–27.

PARNAS, D. L. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM 15,* 12, 1053–8.

PARNAS, D. L. 1979. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering SE-5,* 2, 128–38.

SANGAL, N., JORDAN, E., SINHA, V., AND JACKSON, D. 2005. Using dependency models to manage complex software architecture. In *Proc. 20th Proceedings of the Conference on Object-oriented Programming Systems, Languages and Applications*. 167–176.

SETHI, K., CAI, Y., WONG, S., GARCIA, A., AND SANT'ANNA, C. 2009. From retrospect to prospect: Assessing modularity and stability from software architecture. In *Proceedings of the 8th Working IEEE/IFIP Conference on Software Architecture (WICSA) and the 3rd European Conference on Software Architecture (ECSA)*. 269–272.

SHAW, M. 1990. Prospects for an engineering discipline of software. *IEEE Software 7,* 6, 15–24.

SINNEMA, M., DEELSTRA, S., NIJHUIS, J., AND BOSCH, J. 2004. Covamof: A framework for modeling variability in software product families. In *Proceedings of the Software Product Line Conference*. Vol. 3154. 197–213.

STAFFORD, J. A. AND WOLF, A. L. 2001. Architecture-level dependence analysis for software systems. *International Journal of Software Engineering and Knowledge Engineering 11,* 4, 431–451.

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

SULLIVAN, K., CAI, Y., HALLEN, B., AND GRISWOLD, W. G. 2001. The structure and value of modularity in software design. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*. Vol. 26. 99–108.

SULLIVAN, K., GRISWOLD, W., SONG, Y., AND ET AL., Y. C. 2005. Information hiding interfaces for aspect-oriented design. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*.

TAYLOR, R. N., MEDVIDOVIC, N., AND DASHOFY, E. 2009. *Software Architecture: Foundations, Theory and Practice*. Wiley.

WONG, S. AND CAI, Y. 2009. Improving the efficiency of dependency analysis in logical models. In *Proc. 24th Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. 173–184.

WONG, S., CAI, Y., KIM, M., AND DALTON, M. 2011. Detecting software modularity violations. In *Proc. 33rd Proceedings of the International Conference on Software Engineering*.

WONG, S., CAI, Y., VALETTO, G., SIMEONOV, G., AND SETHI, K. 2009. Design rule hierarchies and parallelism in software development tasks. In *Proc. 24th Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. 197–208.