Simon: Modeling and Analysis of Design Space Structures

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

The structure of the coupling relation on design decisions is a key factor influencing the evolvability properties and the economic value of a design. The work of Baldwin and Clark is an important step toward a theory of the relationship between structure and value. A key step to enabling rigorous validation and perhaps the eventual use of their ideas for software engineering is formalization of their model. In this paper, we present a brief overview of such a formal model and a prototype software tool, *Simon,* implementing it. We present Simon's functions for deriving design structure matrices and computing impacts of changes in design decisions, and we sketch an initial experimental evaluation in the form of a replication study of our earlier analysis of Parnas's 1972 paper on information hiding modularity.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design

General Terms

Design

Keywords

Design Structure Matrix, Design Rule, Dependence

1. INTRODUCTION

The structure of the coupling relation on design decisions has long been seen as a key determinant of evolvability [15, 13]. More recently, Baldwin and Clark have drawn connections between design decision coupling structure and economic value [2]. A key step to rigorous validation of their work is the formalization their notions of design space, design structure and options value, and provision of prototype tools for building and analyzing such models.

In this paper we sketch our recent work to develop formal models sufficient to capture the core modeling approach of Baldwin and Clark [4]. Our focus to date has been on formalizing the design space concept. Future work will draw links to options value using their valuation methods.

Our work builds on a long tradition of using logical constraints to represent design spaces. It extends the basic approach to account for the representations and reasoning that appear in Baldwin and Clark's work. We are implementing a prototype tool for constructing and analyzing models. We call it *Simon*. It is meant to support further development and evaluation of the underlying design theory and is not presented as having immediate industrial utility. We are addressing scalability issues in a separate line of work centering on the modular analysis of Simon models.

Simon supports interactive construction of formal models, derives and displays design structure matrices (the representation at the heart of Baldwin and Clark's work) and supports simple design impact analysis. We are developing and analyzing models of several published designs, including a study of Parnas's KWIC as it appeared in our own earlier work [18]. Our results generally confirm the conclusions of earlier works, but our results do so rigorously and have generally revealed some errors in those works.

In the next section, we sketch our modeling approach. A longer work [4] provides details. Section 3 presents Simon features and usage. Section 4 discusses related work. Section 5 concludes.

2. DESIGN MODELING AND ANALYSIS

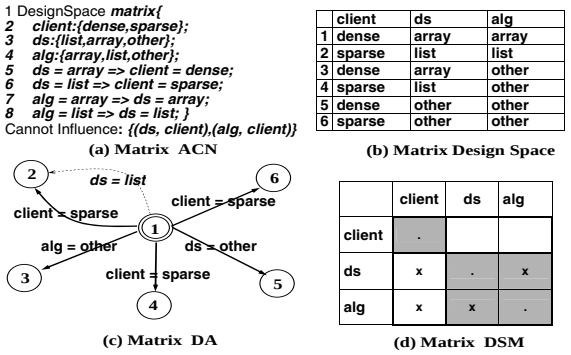
In this section, we present our modeling and analysis approach. We then present a simple example to make the ideas more concrete.

2.1 Introduction to Our Modeling Approach

In this, our initial attempt at a formal theory, we model design spaces using finite-domain constraint networks (CNs) as the key element [12]. To account for the work of Baldwin and Clark, we found we needed to augment such models to express ex ante clustering of variables (e.g., into architecture and implementation decisions, respectively), and to model what we call a *cannot-influence* relation, which states whether changes to one variable can force changes to another. We call our models *augmented constraint networks* (ACNs).

From an ACN we derive what we call a Design Automaton (DA). The states of a DA model all of the designs in the design space, as bindings of variables to values satisfying all constraints. An arc, labeled by a change to one variable, models a transition to a new design state in which a minimal

ASE'05, November 7–11, 2005, Long Beach, California, USA.



	client	ds	alg
1	dense	array	array
2	sparse	list	list
3	dense	array	other
4	sparse	list	other
5	dense	other	other
6	sparse	other	other

	client	ds	alg
client	.		
ds	x	.	x
alg	x	x	.

Figure 1: Matrix Example

set of changes is made to other variables as needed to restore consistency. ACNs are often non-deterministic. Because they encode all ripple effects of change, they allows us to formulate and solve a variety of *design impact analysis* (DIA) problems.

The central design space modeling notation in the work of Baldwin and Clark is the *design structure matrix*, developed by Steward [17] and elaborated by Eppinger [5] and others. A DSM is a matrix with design variables on the rows and columns and simple marks in the cells to mark dependences between the designated design decisions. Modularity is formulated as a specific block-diagonal structure in the markings of a DSM.

Although DSMs seem easy to use and understand, they are inadequate as a basis for a formal theory of the value of modularity in design. DSMs do not represent the values possible for each variable, nor the constraints that create dependences, so they have no clear semantics. They also model only pair-wise dependences. Our work starts by modeling design spaces at the more abstract, semantically rich level of ACNs, and we then derive DSMs from the DAs implied by such ACNs.

We now present additional details through a simple example involving decisions made in designing a *matrix* class, e.g., for a numerical analysis package. This design comprises two variables: the choice of a data structure and of an algorithm operating on it. The three data structure choices currently recognized are array, list and some other—as yet unelaborated—possibility. These choices are in turn constrained by an environmental variable outside the designer's control [18]: whether the matrix client needs dense or sparse matrices. Arrays are best for dense matrices; and linked lists, for sparse matrices.

2.2 Modeling Approach

Figure 1 (a) presents our model of this design space. It contains three variables (Lines 2 to 4) modeling client needs and data structure and algorithm choices. The domains (possible choices) are given in curly braces. The constraints indicate that the choice of an algorithm depends on a choice of data structure, and that that choice in turn depends on the client's needs. We model these relations as implications, depicted on lines 5–8 of Figure 1 (a). Figure 1 (b) presents the solutions of the CN, each of which models a valid *design state*.

The central concept in the work of Baldwin and Clark is that of the *design rule*. A design rule is a design decision that decouples other design decisions by imposing certain constraints—an interface—on them. Accounting for the notion of design rules presented a challenge to our modeling framework. Consider the simple example of a constraint $a = b$. A change in a could be accommodated by one in b, and a change in b by one in a. If, however, a is a design rule, then a can be changed, forcing a change in b, but b cannot be changed in a way that forces a change in a. Architectural design decisions often dominate implementation decisions in just this way (although in practice, dominance is often mitigated by negotiation).

We contribute the notion of a binary relation on design variables called *cannot-influence*. Here, *b cannot-influence a*. In our matrix example, client preference dominates the design decisions. The designer cannot force changes in client preferences.

We also needed to represent ex ante clustering of design variables into rough proto-modules: e.g., into environment, architectural and detailed design decisions. We do not discuss clustering in any detail in this paper.

2.3 Analysis Techniques

Figure 1 (c) presents part of the DA for our example, starting from an arbitrary labeled state 1: a state with a client needing dense matrices and with an array data structure. Changing the client preference to sparse makes the design inconsistent. Making a set of minimal changes to other variables to restore consistency leads to state 2, 4, or 6. The *cannot_influence* relation precludes some *logically* valid transitions. In Figure 1 (c), because *(ds, client)* is in *cannot_influence*, the transition triggered by changing *ds* from *array* to *list* and leading to the client change in state 2 (the dotted arrow labeled *ds = list*) is precluded.

Simon provides editors for CNs, *cannot-influence* relations, and ex ante clusterings. Simon works by translating a CN into an Alloy specification [8], invoking the Alloy constraint solver to get the valid design states, and then computing the DA. Given a DA, we can now reason formally about a design space in ways that are either informal or only hinted at in the work of Baldwin and Clark. In this paper, we identify two, and focus on the second.

First, what are all of the ways to accommodate a given sequence of changes to design decisions? We formulate the analysis as a mapping from a DA, a current design state, and a sequence of variable-value pairs that model changes, to the set of design states at the ends of feasible paths through the DA for the given changes. Associating estimated costs with design changes would then provide a way to reason backwards from the end of a sequence to estimate which initial change is best for long-term cost reduction.

Second, from a DA we can derive a DSM with formally justified marks. The key is to reduce a DA to a set of pair-wise dependences. We define two variables to be pair-wise dependent if, for some design state in the DA there is some change to the first variable for which there is a minimally perturbed consistent state in which the second is changed. A DSM is computed by marking a matrix with these pair-wise dependences, and then (a topic beyond the scope of this paper) by re-ordering rows and columns to group variables in cohesive clusters. Figure 1 (d) presents the derived DSM for the matrix example.

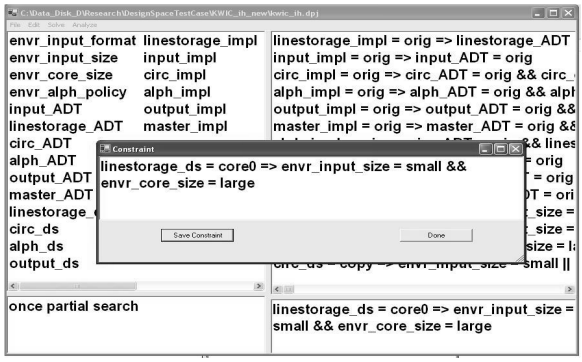


Figure 2: Snapshot: IH Design in the CN GUI

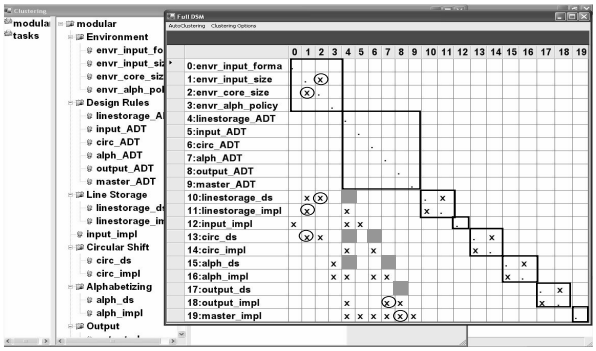


Figure 3: Snapshot: Clustering and DSM GUIs

3. EXPERIMENTAL PROOF OF CONCEPT

We are evaluating Simon by experimental application to a range of modeling and analysis problems. We present one small-scale experiment replicating one we conducted manually in our initial work on testing Baldwin and Clark's ideas for software design [18]. The case is a study of Parnas's seminal formulation of information hiding modularity in software design using several designs for a Key Word in Context (KWIC) program. The example is not new, but formal modeling and automated analysis in relation to the idea of design rules, as a generalized account of information hiding, does appear to be new.

We model the two design spaces in Parnas's analysis [13]. In the first, sequential design (SD), 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 SD design comprises five broad concerns (see Parnas's paper for details): *Input*, *Circular Shift*, *Alphabetizing*, *Output*, and *Master Control*. The IH design adds another, *Line Storage*. Both design spaces exist in an environment comprising four relevant variables: input size, core size, alphabetizing policy and input format.

Figure 2 presents the IH design. The choices of a line storage abstract data type interface, data structure and a program that manipulates it are modeled by design variables, starting with *linestorage*, in the upper left box. The lower left box presents the domain (set of modeled values) for the selected variable *envr_alph_policy*. The right upper box shows the constraints entered through the popup window in which a constraint is being defined: if the input size is small and core size is large, then no packing is needed.

The *cannot-influence* editor presents a matrix labeled with CN variables. Marking a cell states that variable on a given row cannot influence the one on the given column. For lack of space we omit any illustration.

Figure 3 shows part of the cluster editor with a list displaying two methods that can be used to automatically re-order rows and columns into clusters. The list besides it shows the hierarchical structure of the selected clustering. After solving the ACN by clicking the "Solve" menu item, the user can open a DSM view from the clustering view, and generate a DSM organized according to the selected clustering method.

The computed DSM is largely consistent with the DSM that we published previously [18], validating the modeling and analysis concept. The differences, however, reveal subtle errors in the earlier DSMs, primarily ripple effects that we had overlooked, suggesting that the automated approach is superior in terms of modeling reliability. The circled cells in Figure 3 represent the dependences that were missing from our manual model. The dark cells show marks that should not have been present in the manual version. We removed a variable, *input_ds*, as redundant with *linestorage_ds*.

Parnas compared the two design space structures by asking the following question: given an original design, and given changes in 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? The number of modules that has to change is an unrealistically simple proxy for the real cost of a change. We take it as a starting point for more elaborate models.

Simon's DIA function, depicted in Figure 4, supports such analysis. The left and middle upper lists show the original IH design and user-specified changes. The middle lower and the right lists show the computed results: only design 2476 can accommodate a change of the input size to large, and two other design decisions also have to be changed to reach this state. Figure 5 shows part of the SD and IH DAs that

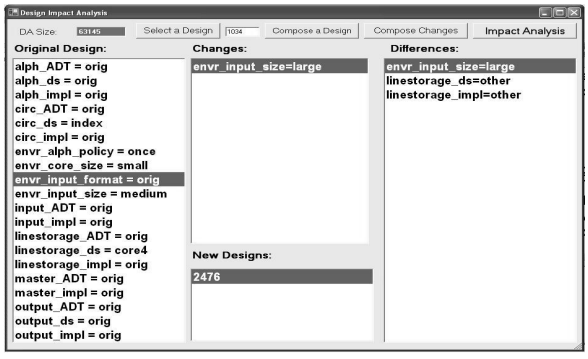
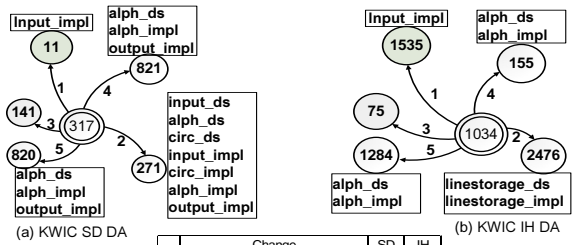


Figure 4: Snapshot: Design Impact Analysis GUI

Simon generated. The numbers in the circles represent the design states of the DAs. The double circles are the start states. Transitions are labeled with changes shown in the table below. The numbers in the last two columns represent the number of 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.



	Change	SD	IH
1	envr input format = new	1	1
2	envr input size = large	7	2
3	envr input size = small	0	0
4	envr alph policy=partial	3	2
5	envr alph policy=search	3	2

Figure 5: Partial DAs for SD and IH designs

4. RELATED WORK

The DSM is central to Baldwin and Clark's account of modularity in design [2]. Its significance for software design was first explored in depth by Sullivan et al. [18, 19] and subsequently by Lopes et al. [10], MacCormack [11], and Sangal et al. [14]. We provide DSMs with rigorous semantics.

Traditional impact analysis tools focus on change at the source code level [1]. Our approach works on abstract but precise design models.

Representing design spaces with constraints is an old idea in Artificial Intelligence [12]. Our DA differs from AI constraint graphs in that we consider more than dependences inferred from constraint syntax. Design space modeling has also been studied by Batory [3], Bosch [16], Lane [9] and Feather [6] for product line design, design generation and optimization. Other formal models include Jackson's Alloy for checking structural properties of designs [8] and Garlan's formalized architectural styles for verifying behaviors [7]. Our purpose and approach differ. We seek to help validate an emerging account of the connections between design structure and value. Our models are thus oriented to formalizing the models of Baldwin and Clark.

5. CONCLUSION

As a first step to bridging the gap between software design and economic analysis, we are developing a design space modeling and analysis approach, and a supporting tool called Simon. A set of small experiments has produced some evidence that our new theory might shed light on poorly understood connections between design structures and economic value. Eventually such analysis might help the practicing designer to make design decisions on economically defensible grounds, even perhaps to make decisions leading to economically better results.

6. ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation under grants ITR-0086003 and FCA-0429786.

7. REFERENCES

[1] R. Arnold and S. Bohner. *Software Change Impact Analysis*. Wiley-IEEE Computer Society Pr, first edition, 1996.

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

[3] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, 1992.

[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] S. D. Eppinger. Model-based approaches to managing concurrent engineering. *Journal of Engineering Design*, 2(4):283–290, 1991.

[6] M. S. Feather. Risk reduction using ddp (defect detection and prevention): Software support and software applications. In *RE*, page 288, 2001.

[7] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development-Volume I*, pages 31–44. Springer-Verlag, 1991.

[8] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[9] T. G. Lane. Studying software architecture through design spaces and rules. Technical Report CMU/SEI-90-TR-18, Carnegie Mellon University, 1990.

[10] 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.

[11] 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*.

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

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

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

[15] H. A. Simon. *The Sciences of the Artificial*. The MIT Press, third edition, 1996.

[16] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. Covamof: A framework for modeling variability in software product families. In *Proceedings of SPLC 2004*, volume 3154, pages 197–213, August 2004.

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

[18] 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.

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