University of Virginia, Department of Computer Science Technical Report CS-2004-19, September 1, 2004.
This paper has been submitted to the 27th International Conference on Software Engineering. Revised on 09/23/2004.

Software Design Spaces: Logical Modeling and Formal Dependence Analysis

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

We lack a useful, formal theory of modularity in abstract software design. A missing key is a framework for the abstract representation of software design spaces that supports analysis of design decision coupling structures. We contribute such a framework. We represent design spaces as constraint networks and develop a concept of design decision coupling based on the minimal change sets *of a variable. This work supports derivation, from logical models, of design structure matrices (DSM's), for which we have a promising but inadequate theory of modularity. We present complexity results and a brute force algorithm. To test for potential software engineering utility, we analyzed the design spaces of Parnas's 1972 information hiding paper, with positive results that were surprising in several ways.*

1. Introduction

The motivation for this work is in the need for a value-based theory of software design [9]. A key source of value is adaptability through modularity in the coupling of design decisions [7]. Today we have no formal, useful theory of modularity in abstract design spaces. The contribution of this work is such a theory based on constraint network (CN) [20] representations of software design spaces and a logical formulation of design decision interdependence.

We use variables to represent dimensions in which design decisions are made, values to represent design decisions, and constraints to model required relations. The set of consistent assignments constitute the design space, and a consistent assignment, a design. The key idea is in the *minimal change sets* (MCS's) for a design decision in a design: the set of minimal subsets of variables that could be changed to restore consistency when the given decision is changed in a given way. Two design variables are defined to be pairwise dependent if, for some design, there is some change to the first for which the second is in some MCS.

In earlier work [32] we showed that Baldwin and Clark's real-options model of the value of modularity shows some promise of contributing to a value-based theory for software design. Following their lead, we represented design spaces as *design structure matrices* (DSM's) [30] [12], with design variables on the rows and columns, and pairwise dependence marks in the cells. The coupling structure is seen in the matrix, with modularity in the form of block-diagonality.

Unfortunately, DSM's are expressively weak. They do not model design choices or the semantics of dependences, and they only model pairwise dependences. It's often hard to know what a mark in a DSM means, or whether one should even be present in a given case. We show that DSM's are lossy summaries of the MCS's of a CN, and that deriving them is NP-complete. We developed a brute-force algorithm, which we used in the experiments described in this paper.

To evaluate our approach for potential engineering utility, we tested its ability to expressively represent and usefully analyze the design space structures in Parnas's information hiding paper [23]. The results were positive and interesting. The rest of this paper is organized as follows. Section 2 reviews DSM's. Section 3 reviews CN modeling in design. Section 4 formalizes our notion of variable dependences in CN's and presents a mapping from CN's to DSM's. Section 5 analyzes the computational complexity of this mapping. Section 6 and 7 present our results. Section 8 discusses related work. Section 9 evaluates this work, including open problems for future research. Section 10 concludes.

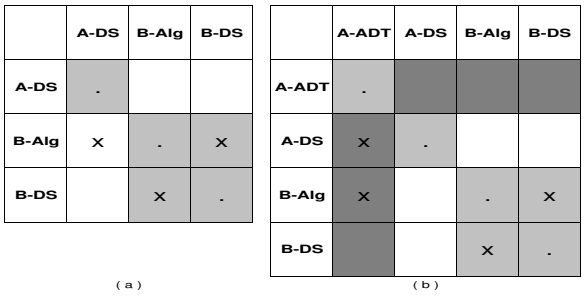


Figure 1. DSM for a design of three variables

2. Design Structure Matrices

The DSM—developed by Steward [30], extended by Eppinger [12], and now supported by tools such as DeMaid [18]—is used for design task structuring and optimization in a range of industries. The DSM is also the basis for Baldwin and Clark's concept of modularity. This section reviews the DSM and its role in the that conception.

Figure 1 presents an example. The design variables *A-DS*, *B-Alg* and *B-DS*, represent choices of values for the data structure for some program A, and two dimensions for a program, B: its algorithm and its data structure. Marks in a row show what decisions a given decision depends upon. DSM's represent design spaces. A set of design variables can be clustered as a *proto-module*: a set of decisions to be made collectively. A proto-module is in essence a composite variable. In Figure 1 (a), gray blocks denote proto-modules. Algorithm and data structure choices for B are made as one, for example.

Decisions made in collaboration are said to be *interdependent*. *Hierarchical dependence* arises where one decision influences but is not influenced by another. *Independent* decisions can be made without communication. Algorithm and data structure choices are often interdependent. The symmetric marks in row *B-Alg*, column *B-DS* and in the symmetric cell, *(B-DS, B-Alg)*, in the first DSM model this condition: an efficacious choice of either depends on the other. The X in *(B-Alg, A-DS)* models a dependence of B's algorithm on A's data structure, as might arise in structured design. The lack of symmetry indicates a hierarchical dependence: A's choice of data structure dominates B's choice of algorithm.

Whereas a proto-module can depend on other variables (as B depends on A), a true module, in Baldwin and Clark's view, is independent, with no marks in the rows or columns outside its bounding box, except as explained below. To obtain a modular design, any dependencies across proto-modules must be broken. In Figure 1 (b) the dependence of *B-Alg* on *A-DS* is eliminated in favor of a hierarchical dependence of A and B on a new decision, *A-ADT*: the choice of a shared interface. Here we have the essential concept of modularity in Baldwin and Clark's work: *independence modulo design rules*. A design rule is a decision that serves to decouple others decisions. Dark shading of the *A-ADT* variable in Figure 1 (b) denotes that *A-ADT* is a design rule for this design. Modularity appears as block-diagonality in a submatrix governed by a set of design rules.

3. Modeling Design Spaces with Constraints

In our earlier work we developed DSM's manually and it was hard to be sure they were valid. We now introduce constraint networks (CN's) as a far more expressive modeling notation. Using CN's in design is not new. Typical applications use constraint solvers to find designs under complex constraints. Our goal, by contrast, is a new coupling theory. Subsection 3.1 develops the idea informally. Subsection 3.2 uses Z [26] to formalize the CN model we use. Subsection 3.3 explains how we represent design spaces as CN's. Subsection 3.4 describes how we represent them in Alloy [17], in particular.

3.1. Informal Introduction

Software design spaces are naturally representable in formal, abstract terms as CN's. Variables represent dimensions of design decisions, sometimes called *concerns* [11]. The domain of a variable represents possible design choices. Bindings reflect actual choices. Variables can also represent conditions in the *environment* that affect design decisions [32]. Constraints express arbitrarily complex dependences among design and environment variables.

We conjecture that a key advantage of logical models is that any concern can naturally and separately be represented as a variable: from choice of a real time response time, to logging or security policy, to type signature, data structure, algorithm, or code body. Separation of concerns, a seemingly complex issue requiring aspects [19] in implementation, becomes a choice of a set of variables in logical modeling. Here, we focus on kinds of choices faced in object-oriented design, deferring exploration of the broader potential to future work.

Consider a simple design space comprising the choice of a function signature and implementation. A CN model could include two variables: *signature* and *implementation*. Their domains represent the choices under consideration. We could include a special *unknown* value to model possible but as yet unelaborated choices. Suppose a designer has developed a particular signature and an implementation, modeled by values *sig_1* and *impl_1*. The domain of *signature* is {*sig_1, unknown*}; and, of *implementation*, {*impl_1, unknown*}.

A key constraint is that the choice of implementation assumes a choice of a signature. We can model this relation using *implication*. The binding of the assuming variable implies the assumed binding: *implementation = impl_1* $\Rightarrow$ *signature = sig_1*. This model captures the idea that changing the signature can forces a change in implementation, but changing the implementation doesn't necessarily force a change in signature. If *implementation* is *unknown*, for instance, *signature* can take any value of its domain.

In his information hiding paper [23], Parnas informally framed design decisions, environment conditions, and constraints among them. For example, when describing an input module, he said, "This module reads the data lines from the input medium and stores them in core for processing by the remaining modules. The characters are packed four to a word…". We find an important design variable and one of its possible values: the input data structure specification, and a choice to pack four to a word. We can formalize this idea using a design variable *input_data_structure* with *packed* as one of its values.

Similarly, his statements on the circular shift module, "…It prepares an index…It leave its output in core…," can be represented by a design variable representing the choice of circular shift data structure, and a particular value, index in core, modeled by the variable *circ_data_structure* and the value *index*.

Variables can also represent aspects of the environment that drive design decisions. One of the main change drivers in Parnas's case study is the size of inputs in relation to memory. Parnas makes several points. First, in his original design, input characters were packed four to a word, so that the input would fit in memory. Second, Parnas notes that "[i]n cases where we are working with small amounts of data it may prove undesirable to pack the characters…." Third, he observes that, "For large jobs it may prove inconvenient or impractical to keep all the lines in core…." In this case, data will be stored on disk.

We can now formalize this description, making a few assumptions where necessary to fill gaps in the informal presentation. The environment variable *envr_core_size* represents memory size, with values $\{$*large_core*$\}$ and $\{$*small_core*$\}$. The environment variable *envr_input_size* represents the input data size, with values *large_input* (too big even for a large memory), *small_input* (fits packed in a small memory or unpacked in a large memory), and *medium_input* (fits in either memory if packed). The design variable *input_data_structure* represents a choice of a input data structure, with values *unpacked*, *packed*, and *disk*. Constraints express the conditions under which various decisions are valid. The effort to store data on disk is worthwhile only for large inputs; the choice to store data unpacked works only for small inputs and large memories; and the choice to pack data makes sense only for small and medium input sizes:

$input\_data\_structure = disk \Rightarrow envr\_input\_size = large\_input;$
$input\_data\_structure = unpacked \Rightarrow envr\_input\_size = small\_input \wedge$
$\quad core\_size = large\_core;$
$input\_data\_structure = packed \Rightarrow envr\_input\_size = small\_input$
$\quad \vee\ envr\_input\_size = medium\_input;$

Parnas also discusses choices for the circular shift store. In his first information hiding design, only indices into input data are stored. In an alternative, copies of shifted lines are stored. Parnas says, "for a small index or a large core, writing them out [copying] may be …preferable [to indexing]…" We thus have another value for variable *circ_data_structure*, *copy*, and a constraint: copying assumes small inputs or a large memory.

$circ\_data\_structure = copy \Rightarrow envr\_core\_size = large\_core$
$\quad \vee\ envr\_input\_size = small\_input;$

It's no surprise that Parnas's informal design space description is incomplete. As stated, it permits at least one set of design decisions that makes little sense: packing inputs and copying shifts in the case of a small input and small core. This problem could be repaired by changing the constraint above to state that copying shifts assumes a large memory, by adding a constraint that copying assumes unpacked inputs, or in other ways.

We add *unknown* to the domains of selected variables to model choices not yet elaborated but that likely exist. The variables and their domains are as follows. The constraints are all given above. We have thus developed a CN model of a small part of Parnas's example.

$input\_data\_structure : \{packed, unpacked, disk, unknown\}$
$circ\_data\_structure : \{index, copy, unknown\}$
$envr\_input\_size : \{small\_input, medium\_input, large\_input, unknown\}$
$envr\_core\_size : \{small\_core, large\_core, unknown\}$

3.2. Finite-domain Constraint Networks

In the design spaces we consider, each choice is made from a finite set. Many designs have this form. We model such spaces as *finite-domain constraint networks* (FDCN's). We now formalize, in Z [27], a cleaned up version of Tsang's formulation of FDCN's [33]. We start with *variables* and their *values*. Each variable takes values from a *domain*, and valid assignments respect these domains.

$[Variable, Value]$

Domains
$domain : Variable \leftrightarrow Value$

Assignment

Domains
$valueof : Variable \pfun Value$

$\forall v : \mathrm{dom}\, valueof \bullet valueof(v) \in domain(\mid \{v\} \mid)$

A set of variables may be subject to a set of constraints. Each such constraint is modeled as a set of assignments: bindings of values to variables that the given constraint allows. The *constraints* relation defined here maps a given subsets of variables to a set of constraints on the variables.

Constraints

$constraints : \mathbb{F}\, Variable \rel \mathbb{F}\, Assignment$

$\forall varset : \mathbb{F}\, Variable;\ asgnset : (\mathbb{F}\, Assignment) \bullet$
$(asgnset \in (constraints(\mid \{varset\} \mid))) \Rightarrow$
$(\forall asgn : asgnset \bullet (\mathrm{dom}\, asgn.valueof = varset))$

A FDCN is a triple (V, D, C), with V, a finite set of variables; D, their *domains*; and C, *constraints* on subsets of V. A *valuation* assigns values to variables respecting domains. The function *valuations* maps a FDCN to its valuations. A valuation *satisfies* a constraint if and only if the valuation is consistent with at least one permitted assignment. A *solution* of a FDCN is a valuation satisfying all constraints. The set of solutions is the *solution space*. The function *solutions* maps a FDCN to its solution space.

ConstraintNetwork

$V : \mathbb{F}\, Variable$
Domains
Constraints

$\mathrm{dom}\, domain = V$
$\mathrm{dom}\, constraints = \mathbb{F}\, V$

$valuations : ConstraintNetwork \fun \mathbb{F}\, Assignment$

$\forall cn : ConstraintNetwork \bullet valuations(cn) =$
$\{asgn : Assignment \mid (\mathrm{dom}\, asgn.valueof) = cn.V\}$

$solutions : ConstraintNetwork \fun \mathbb{F}\, Assignment$

$\forall cn : ConstraintNetwork \bullet solutions(cn) =$
$\{asgn : Assignment \mid asgn \in valuations(cn) \land$
$(\forall asgnset : (\mathrm{ran}\, cn.constraints) \bullet$
$(\exists conasgn : asgnset \bullet conasgn.valueof \subseteq asgn.valueof))\}$

3.3. Modeling Design Spaces with Constraints

We can now formalize the models developed above. The design space for the function example is given by the FDCN that follows. A valid design is modeled as a solution of such a FDCN. A valuation that is not a solution models an invalid design. For example, {*(signature, unknown), (implementation, impl_1)*} is not valid. Designs in our model can have *unknown* values. For example {*(signature, sig_1), (implementation, unknown)*} is valid, as is {*(signature, unknown), (implementation, unknown)*}, which says there are valid combinations of signatures and implementations not yet considered. An infinite set of choices is thus represented by a small, finite model.

$V = \{signature, implementation\};$
$D = \{(signature, sig\_1), (signature, unknown),$
$(implementation, impl\_1), (implementation, unknown)\}.$
$C = \{\{(signature, sig\_1), (implementation, impl\_1)\},$
$\{(signature, sig\_1), (implementation, unknown)\},$
$\{(signature, unknown), (implementation, unknown)\}\}$

3.4. Representing Design Spaces using Alloy

Enumerating the assignments that each constraint permits is impractical. We need to state them declaratively. We chose to use Alloy (Version 2) [16], a first order relational logic, for its balance of expressiveness and tractability and its existing solver. At this stage of research, we didn't require high performance. We now show how we represented our design spaces as FDCN's in Alloy.

In Alloy, *sig* defines a set, and *static part* partitions the set following *extends* into subsets, each of which has only one instance. We use these constructs to model design choices for a variable. The following definition defines the domain for the input data structure specification variable. The other domains are defined similarly.

sig input_DS_spec{}
static part sig unpacked, packed, disk, unknown_input_data_structure
extends input_DS_spec{}

We partition a model into disjoint sets of design and environment variables. The *design* signature specifies two design variables: *input_data_structure* and *circ_data_structure*. The *envr* signature specifies two environment variables: *envr_input_size* and *envr_core_size*. A *model* combines the two. Finally, we represent constraints as Alloy *facts*. Having represented the FDCN, we can use the Alloy solver in computing the solution space.

sig design{input_data_structure : input_DS_spec,
circ_data_structure : circ_DS_spec}
sig envr{envr_input_size : input_size, envr_core_size : core_size}
sig model{e : envr, d : design}

```
fact {
  all s : model | all x : s.d.input_data_structure |
    all y : s.e.envr_input_size |
    x = core4 => y = medium_input or y = small_input
  all s : model | all x : s.d.input_data_structure |
    all y : s.e.envr_input_size | all z : s.e.envr_core_size |
    x = core0 => y = small_input and z = large_core
  all s : model | all x : s.d.input_data_structure |
    all y : s.e.envr_input_size | x = disk => y = large_input
  all s : model | all x : s.d.circ_data_structure |
    all y : s.e.envr_input_size | all z : s.e.envr_core_size |
    x = copy => y = small_input or z = large_core }
```

4. Formal Dependence Analysis

Changing a design decision can violate given constraints. In our function example, starting with the design, {*(signature,sig_1),(implementation, impl_1)*} and changing *signature* to *unknown* violates a constraint, producing an invalid design state. If such an invalidating change must stick, consistency restoration demands changes to some subset of other variables. Here we can change *implementation* to *unknown* (when a function signature is changed, its implementation has to be revisited), and we can say that *implementation depends* on *signature*. Not all changes cause ripples. Starting with {*(signature, sig_1), (implementation, impl_1)*} and changing *implementation* to *unknown*, for example, yields a valid design.

Any sensible definition will define variables to be dependent only if a change to one in some sense *forces* a change in the other. We don't want a definition of dependence that makes every variable depend on every other merely because we *could* change every decision in response to a given change, but only because, in some sense, we *must* change a given variable. The situation is complicated because, in general, there are several ways to compensate for a change. These observations lead to the basis for our definition of pairwise dependence, in the notion of minimal change sets.

The key idea is that to each solution and invalidating change in the value of a variable there corresponds a set of minimal subsets of variables, *minimal change sets* (MCS's), such that consistency can be restored by some changes to all of those variables, but not by changes to any proper subset. For a given design state and invalidating change—for any *invalidating design state transition*—we call the set of MCS's the *MCS group for the transition*. We then define the *MCS set of a variable* as the union of the MCS groups over all invalidating transitions for that variable in the space of valuations. Finally, we will say that one variable depends on another if the former is in some MCS set of the latter.

We formalize the key notions in the functions, *mcsgroup* and *mcsset*. The first maps an FDCN, a solution, one of its variable, and an invalidating new value for the variable—an invalidating design state transition—to the set of MCS's for the transition. The second defines the union operation.

In formalizing these ideas, we use a utility function *subspace*, that maps a given valuation and a set of variables to the set of valuations obtained by allowing the given one to vary on each of the given variables.

$$
\begin{array}{l}
subspace : ConstraintNetwork \times Assignment \times \\
\quad \mathbb{F}\, Variable \rightarrow \mathbb{F}\, Assignment \\
\hline
\forall\, cn : ConstraintNetwork;\ given : Assignment; \\
\quad vs : \mathbb{F}\, Variable;\ gotten : Assignment \bullet \\
\quad gotten \in subspace(cn, given, vs) \Rightarrow \\
\qquad (\forall\, v : cn.V \mid v \notin vs \bullet \\
\qquad\quad gotten.valueof(v) = given.valueof(v)) \wedge \\
\qquad\quad given \in valuations(cn) \wedge gotten \in valuations(cn) \\
\qquad\quad \wedge\ vs \subseteq cn.V
\end{array}
$$

The key function is *mcsgroup*. Here, the conditions state that changing the value of *x* to *v* invalidates the design, *sol*, resulting in a valuation *notsol*. Among all the valuations that differ from *notsol* in the values of all variables in a *MCS*, there must be at least one solution. None of the valuations that differ from *notsol* only in the values of any proper subset of the *MCS* constitute a valid solution.

$$
\begin{array}{l}
mcsgroup : (ConstraintNetwork \times Variable \times Assignment \times Value) \nrightarrow \\
\qquad (\mathbb{F}(\mathbb{F}\, Variable)) \\
\hline
\forall\, cn : ConstraintNetwork;\ x : Variable;\ sol : Assignment;\ v : Value \bullet \\
\quad (sol \in solutions(cn)) \wedge (x \in cn.V) \wedge \\
\qquad (v \in cn.domain(|\ \{x\}\ |)) \wedge (sol.valueof(x) \neq v) \wedge \\
\qquad mcsgroup(cn, x, sol, v) = \{mcs : \mathbb{F}\, Variable \mid \\
\qquad (\exists\, notsol : (valuations(cn) \setminus solutions(cn)) \bullet \\
\qquad (notsol.valueof(x) = v) \wedge (\forall\, y : cn.V \setminus \{x\} \bullet \\
\qquad notsol.valueof(y) = sol.valueof(y)) \wedge \\
\qquad ((subspace(cn, notsol, mcs) \cap solutions(cn)) \neq \emptyset) \wedge \\
\qquad (\forall\, submcs : \mathbb{F}\, Variable \mid submcs \subset mcs \bullet \\
\qquad\quad (subspace(cn, notsol, submcs) \cap solutions(cn)) = \emptyset)) \\
\quad \}
\end{array}
$$

$$
\begin{array}{l}
mcsset : (Variable \times ConstraintNetwork) \nrightarrow \\
\qquad (\mathbb{F}(\mathbb{F}\, Variable)) \\
\hline
\forall\, x : Variable;\ cn : ConstraintNetwork \bullet \\
\quad mcsset(x, cn) = \{amcs : \mathbb{F}\, Variable \mid \\
\qquad (\exists\, sol : solutions(cn);\ v : cn.domain(|\ \{x\}\ |) \bullet \\
\qquad\quad amcs \in mcsgroup(cn, x, sol, v)) \\
\quad \}
\end{array}
$$

We can now formalize design variable dependence for a FDCN, and thus, in essence, the DSM for a FDCN, as a binary relation on variables, as described above. The function *dsm* maps a given FDCN to the required binary relation: *(x, y)* belongs to the DSM of a FDCN if and only if *y* is present in some element of the MCS set of *x*.

$$dsm : ConstraintNetwork \rightarrow (Variable \leftrightarrow Variable)$$

$$\begin{array}{l}(\forall cn : ConstraintNetwork;\ deps : Variable \leftrightarrow Variable \bullet \\ \quad dsm(cn) = deps \Leftrightarrow \\ \quad ((\mathrm{dom}\, deps \subseteq cn.V) \wedge \\ \quad (\mathrm{ran}\, deps \subseteq cn.V) \wedge \\ \quad (\forall x : \mathrm{dom}\, deps \bullet (\forall y : \mathrm{ran}\, deps \bullet \\ \quad\quad (y \in (deps(\mid \{x\} \mid)) \Leftrightarrow \\ \quad\quad\quad (\exists mcs : mcsset(x, cn) \bullet y \in mcs))))))\end{array}$$

5. The Complexity of Dependence Analysis

A basic question to be answered by a satisfactory coupling theory is, *what is the computational complexity of the required analysis?* We contribute a proof that the problem of computing DSM's from FDCN's is NP-complete (NPC).

Space limitations dictate that we give only a proof sketch. The proof is by reduction from *FDCN satisfiability (FCSP)*, deciding whether a FDCN has a solution, which is NPC [13]. We modify a given FDCN so that its dependence relation is non-empty if and only if the FDCN is satisfiable. The idea is to add a new variable and an equality constraint with an existing variable such that the variables are dependent if and only if the original FDCN is satisfiable.

The conclusion is that reasoning about coupling (and so modularity) in logical design models is intractable.[1] There is no really scalable solution, but this is not to say that there are no useful algorithms. As a first step in exploring the automated application of our model, we have implemented a brute-force algorithm to compute DSM's from FDCN's.

The algorithm directly implements the specification. The first step is to compute the solution space. We use Alloy [17] [16]. The rest of the computation takes the solution space as input. We omit the details for lack of space.

6. Case Study: Parnas's KWIC

As an early test of the claims that our approach promises to support expressive modeling and useful analysis for software engineering, we tested it on Parnas's analysis of two design spaces for the *key word in context* (KWIC) problem [23]. Parnas presents two design spaces: a structured design (SD) in which proto-modules embody steps in transforming input to output, and an information hiding (IH) design, in which interfaces decouple design decisions—modules—that should be able to change independently. Parnas presents a comparative design analysis, postulating likely changes in environment variables and assessing their ripple effects in the respective design spaces.

[1] If the constraints in a notation as expressive as first-order logic with arithmetic, as might be used to model design spaces where resource consumption is critical, it is likely the problem is unsolvable, e.g., by reduction from the unsolvable problem of deciding whether a variable value is forced to zero [10]. If true, there is no effective procedure for reasoning about coupling in logical models.

We developed complete CN models for each design and environment space and then used our analysis method to compute DSM's. The KWIC example is particular useful because, as the subject of our earlier, informal work on DSM modeling and value-based reasoning [32], it provides a baseline for evaluating the current approach.

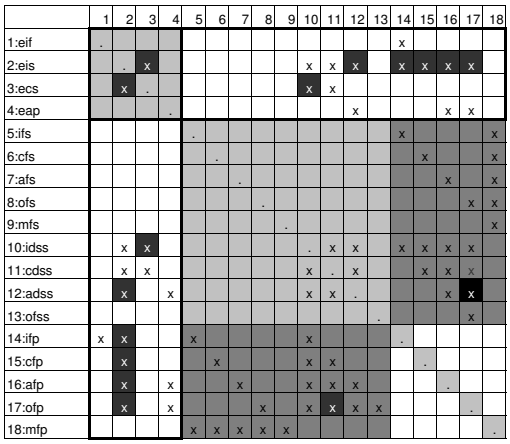
6.1. Modeling the KWIC Design Spaces

For the SD, Parnas describes five modules: Input, Circular Shift, Alphabetizing, Output, and Master Control. Parnas viewed each interface as providing two parts: an exported data structure and a function signature to be invoked by Master Control. Given choices for these parameters, programmers produce function implementations. We modeled the choices of function signature, data structure, and implementation as design variables. Variables *input_fun_sig (ifs)*, *circ_fun_sig (cfs)*, *alph_fun_sig (afs)*, *output_fun_sig (ofs)*, and *master_fun_sig (mfs)* model the function signatures. The choices of implementation are modeled by the variables *input_fun_impl (ifp)*, *circ_fun_impl (cfp)*, *alph_fun_impl (afp)*, *output_fun_impl (ofp)* and *master_fun_impl (mfp)*. Finally, the choices of data structures are modeled by the variables *input_data_structure (idss)*, *circ_data_structure (cdss)*, *alph_data_structure (adss)* and *output_format_data_structure (ofss)*. We analyzed the first two data structure variables in the previous section. The short forms of variable names are used to conserve space in our DSM figures.

In the IH design, a new module, Line Storage, is present. Its data structure variable *linestorage_data_structure* replaces the *input_data_structure* of the sequential design. The IH input module has no separate data structure. In the IH design, each module is also equipped with an abstract data type interface, the choice of which we also model in the standard way. As the result, the variables modeling interface choices are: *linestorage_ADT (ladt)*, *input_ADT (iadt)*,*circ_ADT (cadt)*, *alph_ADT (aadt)*, *output_ADT (oadt)*, and *master_ADT (madt)*. Those modeling implementation choices are *linestorage_impl (lp)*, *input_impl (ip)*, *circ_impl (cp)*, *alph_impl (ap)*, *output_impl (op)*, and *master_impl (mp)*. Finally, those that model data structure choices remain *linestorage_data_structure (ldss)*, *circ_data_structure (cdss)*, *alph_data_structure (adss)* and *output_format_data_structure (ofss)*.

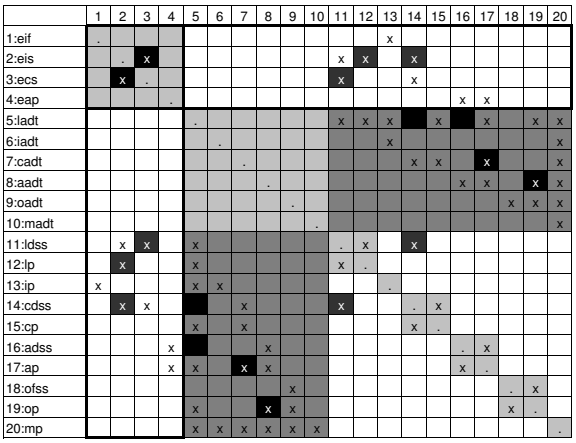
We extend each design variable domain with *unknown* to permit the solver to find designs using unelaborated new values. Parnas assumes original designs in each case and studies the impact of change. We use value *orig* in most of the domains to model Parnas's original choices. For example, the *input_fun_sig* will have domain $\{orig, unknown\}$.

We encoded the design constraints based as closely as possible on Parnas's presentation, in the style described ear-



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1:eif	.													x				
2:eis		.	x							x	x	x		x	x	x	x	
3:ecs		x	.							x	x							
4:eap				.								x						
5:ifs					.									x				x
6:cfs						.									x			x
7:afs							.									x		x
8:ofs								.									x	x
9:mfs									.									x
10:idss		x	x							.	x	x		x	x	x	x	
11:cdss		x	x							x	.	x			x	x	x	
12:adss		x		x						x	x	.				x	x	
13:ofss													.				x	
14:ifp	x	x			x					x				.				
15:cfp		x				x				x	x				.			
16:afp		x		x			x			x	x	x				.		
17:ofp		x		x				x		x	x	x	x				.	
18:mfp					x	x	x	x	x									.

Figure 2. Structured Design Space



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1:eif	.												x							
2:eis		.	x								x	x		x						
3:ecs		x	.								x			x						
4:eap				.												x	x			
5:ladt					.						x	x	x		x		x		x	x
6:iadt						.							x							x
7:cadt							.							x	x		x			x
8:aadt								.								x	x		x	x
9:oadt									.									x	x	x
10:madt										.										x
11:ldss		x	x		x						.	x		x						
12:lp		x			x						x	.								
13:ip	x				x	x							.							
14:cdss		x	x				x				x			.	x					
15:cp					x		x							x	.					
16:adss				x				x								.	x			
17:ap				x	x		x	x								x	.			
18:ofss									x									.	x	
19:op					x			x	x										x	.
20:mp					x	x	x	x	x	x										.

Figure 3. Information Hiding Design Space

lier. The available space does not permit a full listing of the resulting CN model.

Parnas evaluated his designs in an environment of change drivers. In addition to *envr_input_size (eis)* and *envr_core_size (ecs)*, above, we use *envr_input_format (eif)* and *envr_alph_policy (eap)* to model the environment's choices of input format and alphabetizing policy.

6.2. Computing the DSM Models

We present our computed DSM's for the SD and IH design spaces in Figures 3 and 2, respectively. The shading distinguishes kinds of variables. In each DSM, variables 1–4 are environment variables. The gray submatrix in the upper left marks dependences among these variables. The next run of variables in each DSM are the design rule variables. The final run models the remaining open design choices.

Consider the SD case. Parnas noted, "All of the interfaces between the four modules must be specified before work could begin.". These are the choices of function signatures (5–9) and data structures (10–13). The gray submatrix marks dependences among these choices. The dark submatrix beneath marks how changes in design rules impact the remaining variables (14–18): implementation choices made by the programmers. We see that this design is indeed modular: relative to the design rules, the implementation choices are independent, as shown by the lower right block-diagonal submatrix.

In the IH design space, the design rules (5–10) model choices of abstract data type interfaces. These design rules serve to (mostly) decouple independent pairs of interdependent decisions of data structures and procedure implementations (11–20), as seen in the block-diagonal submatrix in the lower right.

7. Experimental Results

Our experiment with KWIC produced results in three basic dimensions: semantic validation of the modeling and analysis technique, including identification of some unresolved issues; the value of intermediate results (minimal change sets); and performance and scalability data.

7.1. Semantic Validation

To evaluate the generated DSM's, we compared them with the ones we produced manually in our earlier work [32]. Overall, our computed DSM's are largely consistent with the earlier results, validating the modeling and analysis concept. There are differences, which we now address.

First, the computed DSM's revealed subtle conceptual errors in our manually produced DSM's, suggesting that logic modeling and automated analysis is more reliable. Cells with differences are marked in white with black backgrounds. For example, our manual SD DSM had no dependence between *output_fun_impl*, and *circ_data_structure*. Parnas's paper confirms its presence, owing to two constraints:

$$output\_fun\_impl = orig \Rightarrow alph\_data\_structure = orig$$
$$alph\_data\_structure = orig \Rightarrow circ\_data\_structure = index$$

Similarly, cells (17, 7) and (19, 8) in our computed IH DSM (and the symmetric cells) revealed dependences missing from our earlier model. The computed DSM also lacked two dependences that should not have been present in the earlier version. An extra variable, *input_data_structure*, redundant with *linestorage_data_structure*, 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.

Second, our computed DSM's are symmetric, whereas our earlier work modeled hierarchical dependence of modules on design rules by the absence of marks in the dark, upper right submatrices. Our new DSM's are symmetric because logical constraints are symmetric. We lack a way to express *hierarchical dependence*, which we now see as an extra-logical construct. Our DSM's model design processes in which interface changes can be driven by implementation decisions as much as the other way around. In practice, this is often the case, with true design rules arising only when interfaces are immutable. We leave the declarative representation of asymmetric propagation as an issue for future work, with the observation that Borning's notion of read-only constraints [2] provides a possible solution.

Third, our results indicate a surprising non-modularity in the IH design space. The off-diagonal dependence, (14, 11) and (11, 14), between *circ_data_structure* and *linestorage_data_structure* shows these ostensibly independent decisions to be interdependent. This unexpected result was cause for concern. There are three possible explanations: the non-modularity is real but previously unrecognized; our IH logical model is flawed; or the marks are artifacts of some aspect of our analysis method. We investigated and found our logic model to be valid. The explanation is that the design decisions are coupled through the environment, due to symmetry in the following constraints.

$$\begin{aligned}&\textit{input\_data\_structure} = \textit{disk} \Rightarrow \textit{envr\_input\_size} = \textit{large\_input};\\&\textit{linestorage\_data\_structure} = \textit{packed} \Rightarrow \textit{envr\_input\_size} = \textit{small\_input}\\&\quad \vee\ \textit{envr\_input\_size} = \textit{medium\_input};\\&\textit{circ\_data\_structure} = \textit{copy} \Rightarrow \textit{envr\_core\_size} = \textit{large\_core}\\&\quad \vee\ \textit{envr\_input\_size} = \textit{small\_input};\end{aligned}$$

In a nutshell, an invalidating change to a design decision can be compensated by a change in an environment variable that, in turn, demands a change in another design variable. Starting with the design, {*(linestorage_data_structure, packed), (envr_input_size, small_input), (envr_core_size, small_core), (circ_data_structure, copy)*}, and changing *linestorage_data_structure* to *disk*, yields an invalid state. To restore consistency, *envr_input_size* has to change *large_input*; but this violates the last constraint. A remedy is to change *circ_data_structure* to *index*, putting *circ_data_structure* in a MCS of *linestorage_data_structure*, creating a dependence.

Design dependences that arise through mutual dependence on environment variables are meaningful, in general. In practice, environmental conditional can sometimes be adjusted to resolve design problems, with further ripple effects in a design. In such cases, design decisions that appear independent, and designs that appear modular, might not be.

In our case, the dependence arguably is spurious. A decision to store data on disk might be made in response to, but not as a driver of, the environment's choice of the input size. Our inability to express hierarchical dependence—the implementation choice cannot drive the environment choice—can lead to such dependences. What the DSM does show is that there are changes requiring coordinated updates to two design decisions. On the other hand, the designer might learn that a disk-based store is too costly, and ask the customer to accept an initial system taking only smaller inputs. Acceptance could then drive changes elsewhere in the design. We would then have a real dependence between apparently independent *design* variables. There is thus nothing wrong with our modeling approach, but it would be improved by an extension for representing hierarchical dependences.

Finally, we formulated the environment variables somewhat differently in our earlier work. The current model maps Parnas's paper directly. The differences in this dimension do not change our basic results.

7.2. Performance and Scalability

The sequential design model has 18 variables, and the IH model, 20. For SD, it took Alloy about an hour on a Pentium 1.5 GHz, 512 MB PC to find the 12018 solutions and 1.5 hours more on two timeshared, 2-CPU UltraSPARC IIIi running Solaris to compute MCS's (and DSM). For the IH model, Alloy took about three hours to find 34907 solutions. *MCS* computation with our brute-force algorithm took another 21 hours.

7.3. Intermediate Analysis Results

A final point is that our intermediate results—the MCS groups for design transitions and MCS sets summarizing them for variables—are potentially useful. The complete, detailed impact analysis is present only in the MCS groups, which tell us, for a given design state transition our minimal compensation options. A DSM by contrast cannot support anything like this level reasoning about change impact. MCS sets provide summary data between MCS groups and DSM's in level of detail. Here is the MCS set for *linestorage_data_structure* in the IH design, for example. Changing *circ_data_structure* and *envr_input_size* is a minimal remedy for some change, but it's never enough to change *circ_data_structure* alone.

$$\begin{aligned}&\textit{mcsset}(\textit{linestorage\_data\_structure}, \textit{kwic}) = \{\\&\quad \{\textit{linestorage\_impl}\}, \{\textit{envr\_input\_size}\}, \{\textit{envr\_core\_size}\}\\&\quad \{\textit{linestorage\_impl}, \textit{envr\_core\_size}, \textit{envr\_input\_size}\}\\&\quad \{\textit{linestorage\_impl}, \textit{envr\_input\_size}\}, \{\textit{linestorage\_ADT}\}\\&\quad \{\textit{envr\_core\_size}, \textit{envr\_input\_size}\}\\&\quad \{\textit{circ\_data\_structure}, \textit{envr\_input\_size}\}\\&\quad \{\textit{linestorage\_impl}, \textit{envr\_core\_size}\}\\&\quad \{\textit{linestorage\_ADT}, \textit{envr\_input\_size}\}\\&\quad \{\textit{circ\_data\_structure}, \textit{linestorage\_impl}, \textit{envr\_input\_size}\}\}\end{aligned}$$

8. Related Work

The idea that coupling structure is a key to adaptability and value in design is old [3, 7, 6, 25, 29]. Using CN's in design is also familiar, although it's not well developed in software engineering. To our knowledge, a coupling theory for logical design spaces as a basis for a formal account of modularity in design is new. Our work complements Baldwin and Clark's [7] with expressive representations, a formal notion of dependence, and a mapping to their DSM's.

We have provided a precise notion of *impact analysis* for logical design models. Traditional impact analysis research focuses on change issues at program level, as summarized in [5]. Advantages of our approach include a precise semantics of dependence, and the ability to reason about the ripple effects of changes in high-level design decisions.

Batory [8] uses formal models of software design spaces for systems that vary in component implementations. His work aims to support system generation and reuse. Jackson [15] used Alloy for object modeling with the goal of being able automatically to prove properties of given models. Garlan et al. [14, 1] 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 CN's in design space search and optimization for complex embedded system design. The goal is to find good designs under constraints (e.g., [22]). Our aim and contribution, by contrast, is a logical theory of coupling in logical design space models.

Axiomatic design [31] uses matrices to represent relationships between design variables and functional requirements in design space search. It is largely geared to continually varying values in phyiscal system design. The approach is only tangentially related to the DSM's of this work.

Our work is related to work on software architecture [1] [4] [21]. 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 [28] studied architectural dependence analysis for architecture definition languages. Our work, in contrast, is not confined to an architectural level, and is entirely formal.

9. Overall Evaluation

In this section we evaluate the novelty and significance of this work, including shortcomings and remaining problems.

Our contributions appear novel in several dimensions. The first is our formalization of variable dependence in constraint networks. The second is the provision of a formal basis for dependence markings in DSM's. The third is a case study supporting the potential utility of constraint-based approaches in software design. The fourth is a formal model and analysis of Parnas's important but informal work. Finally, we put a new emphasis on the design of design *spaces* and their underlying coupling structures, as opposed to the design of individual points in design space, the focus of most current design methods.

Our work has potential significance in several areas. First, it has potential to support a formal abstract theory of modularity in design, and, eventually, to contribute to a value-based theory of architectural design. Second, it provides insight into the complexity of design: reasoning about coupling in logically expressive models is intractable, in general, and, for models with arithmetic, there is probably no effective procedure for coupling analysis. Third, the MCS concept captures the complex ways in which changes can be accommodated in real systems. Fourth, the work clarifies the nature of DSM's, as very lossy models.

This paper leaves many issues open to future work. We briefly discuss of the most important.

First, we only considered finite domain constraints in a first-order relational logic. The Alloy language provides high expressiveness while retaining some tractability. There are many other possible notations and analysis tools. Key dimensions in which results will differ are expressiveness, complexity, and actual performance. The Alloy analyzer was not designed for the purposes to which we are putting it—we used it as a convenience—and our runtimes show that it will not scale to problems much larger than the ones we present. Trading expressiveness for performance is one way to get scalability to larger models.

The fundamental problem though is that design spaces are exponential in the number of variables. Beyond less expressive notations, we need to find techniques that exploit characteristics of special cases that arise in practice. There are many computationally intractable problems that are nevertheless routinely solved: model checking, fault tree analysis, etc. Exploiting known modularity to break large models into parts for analysis is a possible approach. Another approach is to use abstraction to reduce large problems to smaller ones. There's no solving the problem in general. What we need are techniques that can be useful in practice.

We are not yet willing to give up a notation with quantification. We hope to show that our approach accounts for aspect-oriented modularity as a special case [19]. Quantification appears to be a key in this regard [24].

An orthogonal problem is that design spaces are not static in practices. Designers continually change variable sets, domains, etc. Environments have been developed that support CN evolution. An interesting question is to what extent can our coupling analysis results be updated incrementally as constraint networks evolve.

10. Conclusion

In this paper, we contribute an approach to modeling software design spaces as constraint networks, a formalization of dependence, a formal mapping from declarative CN's to DSM's by way MCS's, and a case study of the ideas applied to Parnas's KWIC design study. We analyzed the complexity of the analysis problem and implemented a brute force algorithm. We found that the ideas contribute to our understanding of modularity, and of coupling structure more generally, and that they have the potential to lead to practical analysis methods and tools.

Our immediate future work will focus on an extension of our model to express hierarchical dependences. We are considering in particular to represent hierarchy in an *influence* relation, I, on variables, where $(x,y) \notin I$ indicates that changes in x cannot be compensated for by changes in y. Second, we plan to investigate more efficient algorithms for mapping solution spaces to MCS groups and dependence relations. Third, we will investigate more scalable analysis techniques, based on modular analysis of models for which we can infer a decomposition into relatively independent parts. Finally, we plan to develop new tools to support our research, its evaluation, and perhaps eventually its application. Once we have a formal, abstract, useful theory of modularity in design, we plan to return to the question of the value of modularity, with a new set of tools for making our ideas precise.

Acknowledgments

This work was supported in part by the National Science Foundation under grant ITR-0086003. We thank Alan Borning for valuable comments on the ideas in this paper. We also thank Vibha Sazawal for her careful reviewing.

References

[1] G. D. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319–64, Oct. 1995.

[2] M. W. Alan Borning and B. Freeman-Benson. Constraint hierarchies. *Lisp and Symbolic Cmoputation: An International Journal*, 5:223–270, 1992.

[3] C. W. Alexander. *Notes on the Synthesis of Form*. Harvard University Press, 1970.

[4] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–49, July 1997.

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

[6] W. Ashby. *Design for a Brain*. John Wiley and Sons, 1952.

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

[8] 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.

[9] B. W. Boehm and K. J. Sullivan. Software economics: a roadmap. In *Proceedings of the conference on The future of Software engineering*, pages 319–343. ACM Press, 2000.

[10] A. Borning. Personal communication with Kevin Sullivan. 2004.

[11] E. W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, 1982.

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

[13] M. R. Garey, D. S. Johnson, M. R. Garey, and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of Np-Completeness*. W.H. Freeman & Company, 1979.

[14] 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.

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

[16] D. Jackson. Micromodels of software: Lightweight modeling and analysis with alloy. Feb. 2002.

[17] D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 62–73. ACM Press, 2001.

[18] R. James L. Demaid/ga - an enhanced design manager's aid for intelligent decomposition. In *Proceedings of 6th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, Seattle, WA, 4-6 Sept. 1996.

[19] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Finland, June 1997. Springer-Verlag.

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

[21] N. Medvidovic and R. N. Taylor. A framework for classifying and comparing architecture description languages. *SIGSOFT Software Engineering Notes*, 22(6):60–76, Nov. 1997.

[22] S. Mohanty, V. K. Prasanna, S. Neema, and J. Davis. Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. *SIGPLAN Not.*, 37(7):18–27, 2002.

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

[24] S. C. Robert E. Filman, Tzilla Elrad and M. Aksit. *Aspect-oriented Software Development*. Addison-Wesley Professional, 2004.

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

[26] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[27] M. Spivey. The fuzz manual. URL: `http://spivey.oriel.ox.ac.uk/~mike/fuzz/`.

[28] J. A. Stafford and A. L. Wolf. Architecture-level dependence analysis for software systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(4):431–451, 2001.

[29] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–39, 1974.

[30] 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.

[31] N. P. Suh. *Axiomatic Design: Advances and Applications*. Oxford University Press, May 2001.

[32] 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.

[33] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Pr., London and San Diego, 1993.