Improving the Efficiency of Dependency Analysis in Logical Decision Models

Sunny Wong, Yuanfang Cai

Department of Computer Science
Drexel University
Philadelphia, PA, USA
$\{$*sunny, yfcai*$\}$*@cs.drexel.edu*

***Abstract*—To address the problem that existing software dependency extraction methods do not work on higher-level software artifacts, do not express decisions explicitly, and do not reveal implicit or indirect dependencies, our recent work explored the possibility of formally defining and automatically deriving a *pairwise dependence relation* from an *augmented constraint networks* (ACN) that models the *assumption* relation among design decisions. The current approach is difficult to scale, requiring constraint solving and solution enumeration. We observe that the assumption relation among design decisions for most software systems can be abstractly modeled using a special form of ACN. For these more restrictive, but highly representative models, we present an $O(n^3)$ algorithm to derive the dependency relation without solving the constraints. We evaluate our approach by computing *design structure matrices* for existing ACNs that model multiple versions of heterogenous real software designs, often reducing the running time from hours to seconds.**

***Keywords*-Augmented Constraint Network; Dependency Analysis**

I. Introduction

Software modular structure, determined by component dependencies, influences the ease of change accommodation [1], communication needs among developers [2], and economic value of the software [3]. Large-scale software dependency structures are often extracted from source code using reverse engineering tools, such as Lattix[1]. However, it is recognized that direct syntactic dependencies are insufficient for understanding or analyzing software modular structure. And more precise, logical or indirect, dependencies cannot be easily discovered from source code [2], [4], [5] because, for example, critical design decisions may be implicit [5], [6]. Additionally, extracting dependencies from source code can only be accomplished in later stages of the software development process.

To address these issues and to improve our ability to analyze the consequences of software design decisions and modular structures, at early stages of the software development process, we developed a formal model, called the *augmented constraint network* (ACN). An ACN models design decisions as first-class members and expresses how decisions make *assumptions* upon each other using logical constraints [5], [7]. Based on ACN modeling, we formally defined the notion of *pairwise dependency relation* (PWDR) among design decisions, and developed a number of design-level automated modularity and changeability analyses, such as *design structure matrix* (DSM) [3] derivation and change impact analysis [5], [7], [8]. ACN modeling has been used to formalize Parnas's information hiding principle [1], [6] and the notion of *design rules* in Baldwin and Clark's modularity theory [3], [5], [7]. It has also been used to automate and quantify Parnas's changeability analysis [1], [5], to quantitatively assess AO vs. OO design alternatives [8], [9], and to check modularity consistency between design structure and implementation structure [10].

Despite the potential utility of ACN modeling shown in recent work, the automated analysis techniques enabled by ACN models rely upon constraint solving, which is difficult to scale, similar to other formal models used in model checking [11] and formal specification [12], [13]. Different from other formal models, such as Alloy [12], that are designed to check software properties such as consistency, ACN modeling focuses on expressing assumption relations among decisions, and aims to reason about dependencies and modularity properties. Addressing the scalability issue in such a model is particularly important because comprehension difficulty and modularity decay become prominent and relevant only when the software is of certain scale. The current approach of deriving the pairwise dependency structure from an ACN (i.e. the automatic derivation of design structure matrices) requires not only finding one satisfying solution to the constraint network but enumerating all the solutions, making the approach even harder to scale.

To make the model and the analysis techniques applicable to real software systems, we identify some common characteristics possessed by the majority of existing ACN models. These constraint-based formal models represent medium to large scale real software systems, and most of them can be automatically derived [14], [15] from prevailing software models, such as the *unified modeling language* (UML). Leveraging the formalized notion of *design rules* [3] (defined as stable dominating design decisions that decouple otherwise coupled subordinating decisions), we contribute an approach to generating dependency relations from a class of

[1]http://www.lattix.com

more restricted, but highly representative, ACNs with $O(n^3)$ running time, reducing the complexity for these restricted formal models from NP-complete to polynomial.

Our approach is based on the following observations. When investigating the dependency and hence modular structure of a software system, especially for large-scale software systems, we mainly care about whether one dimension makes assumptions about other dimensions, but not *how*. As a result, a variable in a software ACN often has only two values: *orig* to model a current decision, and *other* to model an unknown possible decision that is different from the current one. This simplification makes ACNs different from other formal models that detail the states of each component. Second, the major relation we care about is the *assumption* relation that can be expressed using logical implications.

We call an ACN that exhibits these two characteristics as a *binary augmented constraint network* (BACN, pronounced *bacon*). In this paper, we present an algorithm to derive pairwise dependency relations for BACNs without solving the constraint networks. These two characteristics allow us to automatically derive the *assumption* dependency relation from UML class diagrams or component diagrams of real and large-scale software systems.

It is worth noting that the number of dependencies derived from an ACN transformed from a UML class diagram is much larger than the number of dependencies discovered from corresponding source code using reverse-engineering tools. For example, we identified 622 pairs of dependencies from a UML-transformed ACN for the Minos system to be described in Section V, while using Lattix, only 271 dependencies were identified. The majority of the differences are due to the fact that ACNs makes implicit and indirect dependencies explicit. (The automatic UML to ACN transformation and the potential utility of these implicit and indirect dependencies are outside the scope of this paper.)

In this paper, we focus on addressing the scalability issue for large ACN models. To solve a constraint network with hundreds or thousands of variables is impractical. Nevertheless, software ACN models with this size are often BACN. We studied 61 ACNs, acquired from published work or from on-going projects in modeling real software systems, that model multiple versions of heterogenous software systems. Some of these ACNs were manually constructed, while others were automatically derived from other design models. Of these 61 ACNs, 59 of them shared the two BACN characteristics described above. The two remaining ACNs were of very small scale, with less than 20 variables.

Our evaluation aims to assess whether our approach can generate pairwise dependencies correctly, and whether it can be applied to real, large-scale software systems. We compute the dependency relations, hence the design structure matrices, from all ACNs available in literature, and also from ACNs automatically extracted from real, medium or large-scale software systems. We compare the DSMs derived from these ACNs and compare the time needed to generate them using our new approach and existing approaches. We show that the DSMs are the same and the time needed often reduces from hours to seconds. The results show that this approach has the potential to make constraint-based design modeling and automated dependency analysis techniques applicable in real software systems.

The rest of this paper is arranged as follows: Section II introduces the background of this work and Section III formalizes BACN. Section IV introduces the polynomial dependency derivation algorithm. Section V presents our evaluation results and Section VI discusses related issues. Section VII presents related work. Section VIII concludes.

II. Background

In this section, we describe the mathematical and theoretical background of our work. In particular, we formalize the *augmented constraint network* (ACN) and *pairwise dependence relation* (PWDR) presented in our previous work [5], [7]. The small example ACN used in this section will also be used in Section IV to illustrate our algorithm. In addition, we provide a proof of the computational complexity for deciding PWDR.

A. Mathematical Formalization of ACN

The augmented constraint network was first introduced by Cai and Sullivan [5], serving as a model for capturing design decisions and the assumptions among those decisions. An ACN consists of three parts: a finite-domain constraint network, a dominance relation, and a cluster set (not discussed in this paper).

A constraint network is a tuple $\langle V, U, d, C \rangle$, where $V = \{v_1, v_2, \ldots, v_n\}$ is a finite set of variables representing design dimensions where decisions are needed. Then $d : V \rightarrow 2^U$ is a mapping of variables to a finite set of valid domain values. Hence, U is the universe of domain values for all variables. Lastly, C is a finite set of constraints upon the variables. Figure 1 shows an example constraint network that models the design of a graph library (simplified for illustration). For example, the first constraint says that the decision to use an adjacency matrix data structure assumes that the client needs dense graphs. We use the term *binding* to refer to specifying the value of a variable in a constraint (e.g. $ds = matrix$, $density \neq dense$).

The dominance relation $D \subseteq V \times V$ models an asymmetric dependency relation among decisions, formalizing Baldwin and Clark's concept of design rules [3]. We define two dominance relation pairs for our graph example: $(ds, density)$ and $(algo, density)$. These pairs indicate that the decisions for what data structure and algorithm to use, cannot influence the client's requirement on the density of graphs to be used. In other words, clients will not change the density of the graphs they use solely because of a developer's affection for a particular data structure or algorithm.

Variables:

$$\begin{aligned} density &: \{dense, sparse\} \\ ds &: \{matrix, list\} \\ algo &: \{matrix, list\} \end{aligned}$$

Constraints:

$$\begin{aligned} ds = matrix &\Rightarrow density = dense \\ ds = list &\Rightarrow density = sparse \\ algo = matrix &\Rightarrow ds = matrix \\ algo = list &\Rightarrow ds = list \end{aligned}$$

Figure 1. Graph Library Constraint Network

We define a solution to a constraint network as a mapping $s : V \to 2^U$ such that all variables are mapped to valid domain values $\forall v \in V\ s(v) \in d(v)$ and all constraints are satisfied. Each solution to a constraint network is a valid design for the software modeled. For any constraint network, we define S as the set of all its solutions. Given two solutions $s, s' \in S$, we use the notation $s - s'$ to represent the set of variables that are assigned different values by the two solutions—or formally, $s - s' = \{v \in V \mid s(v) \neq s'(v)\}$.

From the constraint network and dominance relation we can derive a non-deterministic finite automaton, called a *design automaton* (DA) [5]. The set of solutions S to the constraint network form the states of the DA, and each transition models a change in a design decision. Given an initial design $s \in S$, the transition function $\delta(s, v, u)$ denotes the valid designs resulting from changing variable v to domain value u in s. Since changing v to u may violate some constraints, the value of other variables may need to change to maintain a valid design. However, if a variable v' must change to restore solution satisfiability but v' dominates v (i.e. $(v, v') \in R$, meaning that changes to v cannot force v' to change), then such a change is considered invalid. In addition, transitions only show the destination states that differ minimally from the initial state. Formally, we define $\delta(s, v, u) = \{s' \in S \mid s'(v) = u \wedge (\neg\exists \tilde{s} \in S\ \tilde{s}(v) = u \wedge (s - \tilde{s}) \subset (s - s') \wedge (\forall v' \in (s - s')\ (v', v) \notin D))\}$.

Maintaining satisfiability through minimal perturbation form the concept of pairwise dependence relation (PWDR). Cai and Sullivan [7], [16] define PWDR as a set $P \subseteq V \times V$, such that if $(u, v) \in P$, meaning that v depends on u, then v must be changed in some minimal restoration of consistency to the constraint network, which was broken by a change in u. To formally define PWDR, we first define a mapping $\Delta : S \times V \to 2^S$ to be the set of states directly reachable from an initial state s by changing a variable v to any valid domain value (i.e. $\Delta(s, v) = \bigcup_{u \in d(v) \setminus \{s(v)\}} \delta(s, v, u)$). From this, we formally define the PWDR set P such that $(v, v') \in P$ if and only if $\exists s, s' \in S\ s' \in \Delta(s, v) \wedge v' \in (s - s')$.

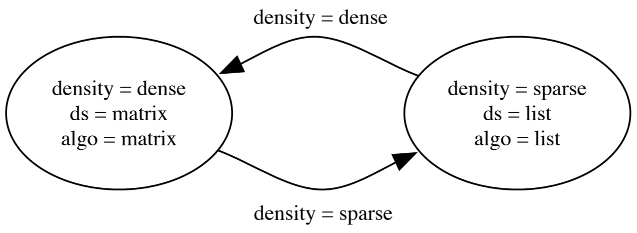

Figure 2. Design Automaton

Figure 2 shows the DA for the small graph library example. Since there are only two satisfying solutions to the constraint network, the DA contains only two states. For simplicity, we refer to the states as *dense* if (*density* = *dense*, *ds* = *matrix*, *algo* = *matrix*), and *sparse* if (*density* = *sparse*, *ds* = *list*, *algo* = *list*). There is only one valid transition from *dense* to *sparse* (by changing *density* to *sparse*) because changing either *ds* or *algo* would require changing *density*, which the dominance relation forbids. Since changing *density* to *sparse* requires changing both *ds* and *algo* to maintain satisfiability, both *ds* and *algo* are pairwise dependent on *density*. The marks in the first column of the *design structure matrix* (DSM) in Figure 3 show these dependencies. In a design structure matrix [3], the columns and rows are labeled with design variables. A cell is marked if there is a dependency between the decision on the row to the decision on the column. The remaining marks between *algo* and *ds* are due to the fact that every time one of them changes, the other does also.

	1	**2**	**3**
1. *density*	**.**		
2. *ds*	**X**	**.**	**X**
3. *algo*	**X**	**X**	**.**

Figure 3. Design Structure Matrix

B. Complexity of Dependency Analysis

The problem of deriving a *pairwise dependency relation* from a constraint network is NP-complete in general. We prove this by reduction from the *constraint satisfaction problem* (CSP) [17]. Given a finite set of variables V', a finite domain of values U', and a finite set of constraints C', the CSP problem is to decide whether there exists an assignment of domain values to variables that satisfy all the constraints. We reduce an instance of CSP to a PWDR decision problem: given an ACN and two variables a and b, decide whether there is a pairwise dependence (a, b).

From a CSP instance, we construct an ACN instance by adding two additional variables $V = V' \cup \{\alpha, \beta\}$, two additional domain values $U = U' \cup \{true, false\}$, and an additional constraint $\alpha = true \Leftrightarrow \beta = true$. Since all variables from the CSP instance can be any domain value, we let $\forall v \in V'\ d(v) = U'$. We restrict the domains of α and β to be only $\{true, false\}$. Hence, the added constraint guarantees that α and β will be assigned the same value in all solutions, and changing the value of α in any solution requires changing β to restore satisfiability.

There is a pairwise dependence (α, β) if and only if the CSP instance is satisfiable. Since $C' \subset C$, if there are no solutions to the CSP instance then there are no solutions to the ACN, and hence there is no pairwise dependence. It is easy to see that if there is a solution s to the CSP instance, then there are two corresponded solutions to the ACN, s_0 with $\alpha = \beta = false$ and s_1 with $\alpha = \beta = true$. Since changing α also changes β, s_0 and s_1 are minimally different when changing α. Hence, if the CSP instance is satisfiable then (α, β) is a pairwise dependence. Therefore, computing PWDR is NP-Complete.

To address the scalability issue caused by constraint solving and solution enumeration, Cai and Sullivan presented a divide-and-conquer algorithm [16] to potentially reduce the number of solutions that need be processed at once. Their algorithm splits an ACN into several smaller sub-ACNs that can be solved independently, then merges the DA from each sub-ACN to derive the full DA. Although this splitting technique does indeed reduce the total number of solutions explored and improves the performance from their initial brute-force algorithm [5], it does not change the problem complexity. The number of sub-ACNs that can be decomposed depends on the quality of design rules in the system. If a system is not well modularized, the divide-and-conquer algorithm will potentially generate sub-ACNs with large number of variables and solutions. Solving these large sub-ACNs and combining them together still exhibit exponential complexity.

The polynomial time algorithm presented in Section IV does not require decomposing a large ACN and recomposing results from sub-ACNs, but only applies to ACNs with restricted forms that we introduce in the next section. In Section V, we compare the running time needed for our new algorithm with the time needed for the divide-and-conquer algorithm, showing that the new algorithm performs significantly better.

III. Binary Augmented Constraint Network

The insight to developing our approach is the observation that the majority of ACNs used to model real software systems exhibit two common characteristics that seem to be sufficient for dependency analysis. First, for dependency analysis, we mainly care about whether one decision makes assumptions on another decision, and whether changing the current decision will influence the decisions on other dimensions, but care less about what the *current* or *changed* decisions are. As a result, the domain of a variable in an ACN often can be abstracted as having only two values: *orig* (the current selected choice) and *other* (an unelaborated future choice). The rationale is that given a changed decision, the designer first needs to know what other dimensions will be impacted, but not what the exact new choices are. Modeling a software design this way will not be sufficient to support certain property analysis as supported by other model checking techniques, such as finding compatible states of all components. Instead, our focus is modularity and dependency analysis. Second, constraints in ACNs represent an *assumption* relation, which often can be expressed using the form $a = \alpha \Rightarrow b = \beta$ to mean that the choice for decision a assumes a certain choice for decision b.

We call an ACN that exhibits these characteristics as a *binary augmented constraint network* (BACN, pronounced *bacon*). More precisely, a BACN is an ACN where each variable has a binary domain and each constraint is a mathematical implication of two bindings $a \Rightarrow b$. Constraints of the form $a \vee b$ and $a \Leftrightarrow b$ can be trivially converted to the required form, so they are also considered valid in BACNs.

In Section IV, we show that, unlike the general case of ACNs, computing PWDR for BACNs is not NP-Complete by presenting a polynomial time algorithm. The key to the tractability of this problem is that BACNs have both restricted constraints *and* domain arity. Any CSP can be transformed to one with only binary constraints [18], so only restricting the constraints does not change the problem complexity. Since each variable in a BACN has only two valid domain values, we can consider one of those values to be $true$ and the other to be $false$, and the BACN becomes equivalent to a 2-CNF instance. In fact, the basis of our algorithm is to consider the bindings as literals in a 2-CNF instance, and leverage existing 2-CNF techniques.

Since the constraint network of a BACN is equivalent to a 2-CNF instance, the most obvious algorithm for computing PWDR is to solve for all satisfying solutions and construct the design automaton. At first this seems like a lucrative approach because we can find an initial satisfying solution, in linear time [19], [20], and enumerate the remaining solutions, in polynomial time per solution [21]. However, we run into a *state explosion* problem [22] because there are an exponential number of solutions in the worst case, making this approach infeasible for large models. In contrast, the algorithm we present in the next section does not enumerate all solutions and therefore its running time is independent of the number of solutions.

IV. Algorithm

We use the example ACN from Figure 1 to illustrate our algorithm. The basic idea is to build an *influence graph*, whose edges are potential PWDR pairs; then use a *compatibility graph* [21] of the constraint network to check for the existence of transitions in the DA, in order to verify the potential PWDR pairs. Both the influence graph and compatibility graph are constructed from an *implication graph*, a well-known structure for solving 2-CNF problems.

The first step of our algorithm is to build an *implication graph* [20], [21] to model the constraints. We create two vertices in the implication graph for each variable in the constraint network: one for each domain value. For notational simplicity, we refer to the vertices (by their 2-CNF equivalents) as v and $\bar{v}$ rather than by specific domain values. For each constraint $u \Rightarrow v$, we create two directed edges (u, v) and $(\bar{v}, \bar{u})$. Figure 4 shows the implication graph for our example ACN. For example, the edges $(algo = matrix, ds = matrix)$ and $(algo = list, ds = list)$ are created for the constraint: $algo = matrix \Rightarrow ds = matrix$. We explain the significance of the dashed edges later in this section.

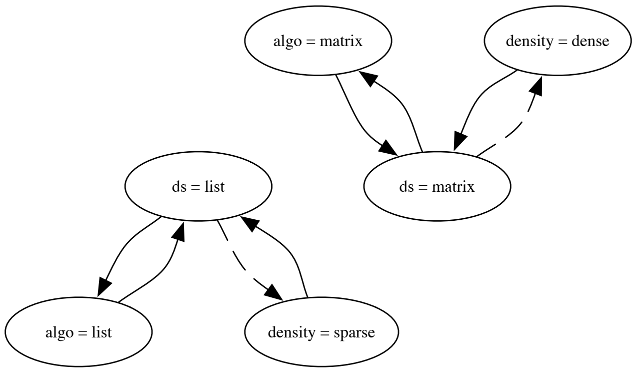

Figure 4. Implication Graph

From the implication graph, we construct a *compatibility graph*, following Feder's algorithm [21] for identifying partial solutions to a 2-CNF instance. A compatibility graph is an undirected graph, with an edge (u, v) if and only if there is an edge $(\bar{u}, v)$ in the transitive closure of the implication graph (not shown). In other words, the edges in a compatibility graph represent the clauses in the transitively closed 2-CNF instance. Figure 5 shows the compatibility graph constructed from the implication graph in Figure 4.

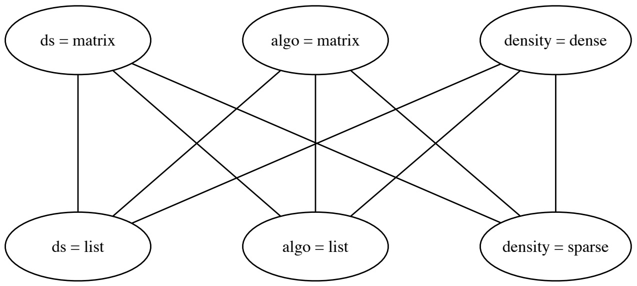

Figure 5. Compatibility Graph

Rather than identifying all solutions to the constraint network and then finding dependent variables from transitions between solutions, we use the compatibility graph to identify partial solutions when checking for a potential dependency between variables. Feder [21] proves that any valid vertex cover to the compatibility graph is a valid partial solution to the 2-CNF instance, and that minimal vertex covers are full solutions. Given a vertex cover S, if both v and $\bar{v}$ are in the cover then v is unassigned in the partial solution; if only $v \in S$ then v is assigned true, and if only $\bar{v} \in S$ then v is assigned false. For example, taking the top vertices in the graph in Figure 5 clearly forms a valid vertex cover and logically is a valid solution/design because it uses adjacency matrices for for dense graphs. Therefore, if we know that changing a variable a may require a change in b to compensate then we can use the compatibility graph to confirm the existence of such partial solutions. We provide a detailed example of this process after defining the influence graph.

In order to account for the dominance relation and identify potential PWDR pairs, we construct an *influence graph* from the implication graph. To construct the influence graph, we first remove the edges from the implication corresponding to the dominance relations. If u cannot influence v (i.e. $(u, v) \in D$) then we remove the edges $(u, v), (u, \bar{v}), (\bar{u}, v)$ and $(\bar{u}, \bar{v})$. The dashed edges in Figure 4 shows the edges that would be removed when constructing the influence graph for our example. We complete construction of the influence graph by collapsing strong components into a single vertex, because all literals in a strong component will be assigned the same value in all solutions of a 2-CNF instance [20], [21]. Figure 6 shows the influence graph constructed from the implication graph in Figure 4. Since $algo = matrix$ and $ds = matrix$ form a strong component in the implication graph, they are merged in the influence graph.

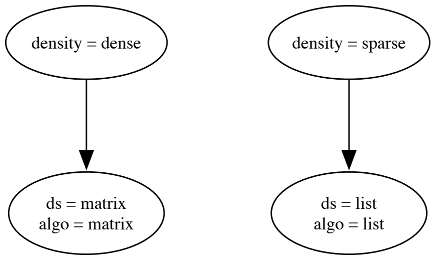

Figure 6. Influence Graph

Each edge (u, v) in the influence graph represents a constraint $u \Rightarrow v$ from the ACN, so that if we change the value of u to $true$ then we may need to change the value of v to $true$ in order to satisfy the constraint. Therefore, we can iterate through each edge (u, v) of the influence graph and use the compatibility graph to identify if v does indeed change to accommodate a change in u in any solution pair.

Since we removed edges based on the dominance relation, we will not consider any edge where we are not allowed to change v in order to compensate for the change in u.

We use the edge $(\{density = dense\}, \{algo = matrix, ds = matrix\})$ from Figure 6 to illustrate this dependence analysis process. Since $algo = matrix$ and $ds = matrix$ will be always have the same value, we arbitrarily pick $algo = matrix$ for checking the compatibility graph. We are only considering $algo$ and $density$, so both vertices for ds are always in our vertex cover. If our potential vertex cover is $S = \{algo = matrix, density = dense, \dots\}$ (i.e. both vertices from the influence graph set as true), then we have a valid vertex cover because there is no edge between $algo = list$ and $density = sparse$ in the compatibility graph. Changing the value of $density$, we would need to complement $algo$ to restore satisfiability so we look at $S' = \{algo = list, density = sparse, \dots\}$. Since S' is also a valid vertex cover, we know that $algo$ changes to compensate for the change in $density$.

Sometimes changing a variable u may cause the constraint to be unsatisfied but there is no way to restore consistency. In such situations, there will be no valid pair of valid vertex covers where both u and v are different values. This scenario often occurs when there is a self-loop in the compatibility graph and hence, a variable remains the same value in all solutions. When such a scenario occurs, we say such an edge is invalid and remove it from the influence graph since it cannot be a PWDR pair.

After removing the invalid edges from the influence graph, we can trivially find PWDR from the transitive closure of the resulting influence graph. The reason we take the transitive closure is because changing a variable may cause a ripple effect. For example, we consider the constraints $a \Rightarrow b$ and $b \Rightarrow c$. If all variables start as $false$, and we change a to $true$, then b needs to change to $true$ to satisfy the first constraint. But changing b to true causes c to change to $true$ to satisfy the second constraint. Since all variables in a strong component are always the same, they are all pairwise dependent. Finally, the set of edges of the transitively closed influence graph makes up the rest of the PWDR pairs.

A. Complexity Analysis

In this subsection, we present the running time complexity of our algorithm. For notation purposes, we let n be the number of variables and m be the number of constraints in the ACN. Our algorithm consists of the following steps and running times:

1) Construct implication graph: $O(n + m)$
2) Construct compatibility graph: $O(n^3)$
    a) Transitive closure of implication graph: $O(n^3)$
    b) Populate edges: $O(n^2)$
3) Construct influence graph: $O(n + m)$
    a) Remove dominance relation edges: $O(m)$
    b) Find strong components [23]: $O(n + m)$
4) Remove invalid edges: $O(m)$
   Since we only need to consider a constant number of edges to verify a valid vertex cover, we spend a constant amount of time per edge.
5) Transitive closure of influence graph: $O(n^3)$

Since $m \leq n^2$, the total running time of our algorithm is $O(n^3)$ polynomial. In addition, we construct three graphs, each with $2n$ vertices and at most $2m$ edges, so our algorithm also uses polynomial space.

V. Evaluation

Given that our purpose is to extract dependency structures, which can be represented using design structure matrices, our evaluation focuses on the effectiveness of generating DSMs from ACNs. The DSM itself has shown to be an effective modularity analysis and visualization model [5]–[7], [24]–[27]. We evaluate the correctness of our algorithm by comparing the DSMs automatically generated using our new algorithm with those generated using the divide-and-conquer approach developed by Cai and Sullivan [16]. We compare the performance of both algorithms in terms of the time needed to generate DSMs from the same set of ACNs. The largest ACN the divide-and-conquer approach can handle, that is, generating DSMs within reasonable time and without running out memory, has only less than 130 variables. To evaluate whether the new algorithm can handle much larger models, we applied the algorithm to ACNs that model real, large-scale open source projects. In this section, we first describe the environment we used for comparing the algorithms, the subject software systems, and the experimental results, highlighting several real software systems we have investigated.

A. Environment

Our experiments use Drexel University's Minos[2] system, a refactored version of Cai's Simon [5], [7] tool. Similar to Simon, Minos allows the user to build ACN models and generate DSMs, but its internal architecture has been significantly improved over Simon. One improvement of Minos is the plugin architecture to support easy feature extension. As a result, both the BACN algorithm and the divide-and-conquer algorithm are implemented in Minos as interchangeable components.

Our experiments ran on a Linux server with two quad-core 1.6Ghz Intel Xeon processors and 4GB of RAM. Minos's implementation of Cai and Sullivan's divide-and-conquer PWDR algorithm also optimizes the running time by parallelizing the construction of DAs using multiple threads. We found, through experimentation, that using four threads in Minos produced the peak performance for their algorithm on our machine. The brute-force algorithm Cai and Sullivan

[2]http://rise.cs.drexel.edu/wiki/Minos

first developed [5] simply enumerated all solutions to the ACN to build a DA, and identified PWDR from the DA transitions. Since this brute-force approach is extremely hard to scale, we do not take it into comparison.

B. Subjects

We studied 61 ACNs in total, modeling both small but canonical software examples widely used in software engineering literature, and medium to large-scale real software systems. Concretely, for small canonical examples, we studied 2 ACNs representing two variations of Parnas's *keyword in context* (KWIC) system [1], [5], [6], 2 ACNs representing AO and OO alternatives of the widely used Figure Editor (FE) example [8], [28], and a MazeGame illustrating design patterns [29]. These 61 ACNs are all BACNs except for the two KWIC models.

We studied ACNs that model real software systems, including 10 ACNs modeling variations of the WinerySearch system studied by Lopes et al. [24] and Cai and Sullivan [16], 6 ACNs abstracting from a fault tree analysis tool called Galileo [7], [30], 3 ACNs modeling the AO, OO, and DR alternatives of a networking system called HyperCast [10], [16], [31], an ACN modeling a financial management system called Vokda [32], and 16 ACNs modeling 8 releases of a software product line called MobileMedia [9], [15], [33] with AO and OO alternatives for each release. The ACNs represent heterogenous real software systems. The commonality among these ACNs are that they are all derived from higher-level software artifacts than source code. The MobileMedia ACNs were automatically generated from UML component diagrams [15], all other ACNs were manually constructed from specifications and design descriptions.

Our purpose is to test whether the algorithm can enable dependency derivation from large-scale models, but the largest ACN model in existing literature only has 81 variables and we do not have large-scale design-level artifacts to transform into ACNs. To test the scalability of our approach, our strategy is to take a real software system of reasonable scale, reverse engineer it into a UML class diagram, and then derive an ACN from the UML class diagram [14]. As we discussed before, significantly larger number of dependencies, mainly implicit or indirect dependencies, can be picked up using this technique than using traditional reverse engineering tools to extract syntactical dependencies. The subjects we selected include the first twelve versions of Hadoop[3], JUnit[4] version 3.4, SAT4J[5], Alloy 4 [12], Kodkod [34], Glusta [10], UML2ACN [14], and Minos itself. Using Minos to generate a DSM modeling itself is an interesting way to assess scalability and modularity of its own. The ACNs generated for these systems are all BACNs, and most have hundreds of variables.

C. Results

In this subsection, we discuss the results of running our algorithm and comparing against the results of Cai and Sullivan's divide-and-conquer algorithm. Given an input ACN, we compute a corresponding DSM using each algorithm and answer the following evaluation questions through the comparison:

1) Does our algorithm produce the same DSMs as Cai and Sullivan's divide-and-conquer algorithm?
2) Does our algorithm outperform Cai and Sullivan's algorithm, in terms of running time?

To answer the first question, we compared all the DSMs generated from BACNs using our algorithm with the DSMs generated using their algorithm. First of all, not all DSMs can be generated using the divide-and-conquer approach because some decomposed sub-ACNs are still too large to be solved within reasonable time without running out of memory. For all the ACNs that can be processed by both algorithms, the DSMs generated are exactly the same, providing a positive answer to the first question.

To answer the second question, Table I lists the running times used to process the ACNs. In Table I, the first column shows the name of the software; the second column shows the number of variables in the ACN; the third column shows the number of constraints in the ACN; the fourth column shows the running time of Cai and Sullivan's divide-and-conquer algorithm [16]; and the last column shows the running time of our BACN algorithm. All algorithm running times are reported in seconds. We use "N/A" to denote when an algorithm takes longer than eight hours to run, or cannot compute PWDR due to memory exhaustion on our machine. Below, we elaborate on the results for several of the larger real systems.

HyperCast: HyperCast [31] is a scalable, self-organizing overlay system developed in Java, with roughly 50 KLOC. Hypercast has been studied in multiple software engineering work [10], [16], [25] for different purposes. Sullivan et al. [25] investigated three different designs of the system (one using object-oriented programming, one using oblivious aspect-oriented programming, and one using design rules and aspect-oriented programming). The DSMs used in this paper were manually constructed. Cai and Sullivan [16] then used ACNs to automatically generate these DSMs, and fixed several errors caused by manual construction in the previous work. Huynh et al. [10] checked the conformance between design and implementation of HyperCast.

The table shows that for all three designs of HyperCast, our algorithm produced the same DSM and finished faster than the divide-and-conquer algorithm. For example, Cai and Sullivan's algorithm took over six minutes to generate the

[3]http://hadoop.apache.org
[4]http://www.junit.org
[5]http://www.sat4j.org

Table I
Running Time Comparison (in seconds)

System	**Vars**	**Consts**	**Div & Conq**	**BACN**
Galileo 1	7	6	1	0
Galileo 2	7	6	2	0
Galileo 3	11	16	1	0
Galileo 4	13	12	2	0
Galileo 5	14	21	1	1
Galileo 6	18	28	1	1
Figure Editor OO	12	9	1	0
Figure Editor AO	13	7	1	0
Winery Locator 1	5	4	1	0
Winery Locator 2	6	6	1	0
Winery Locator 3	12	12	4	1
Winery Locator 4	14	18	4	1
Winery Locator 5	15	8	6	0
Winery Locator 6	23	44	7	1
Winery Locator 7	27	45	4	1
Winery Locator 8	29	13	N/A	0
Winery Locator 9	29	51	2	1
Winery Locator 10	32	55	3	1
Maze Game	24	50	2	2
HyperCast OO	28	42	28	1
HyperCast AO	25	50	368	1
HyperCast DR	33	60	16	1
JUnit 3.4	62	269	6555	4
UML2ACN	19	115	3	1
Glusta	31	188	894	2
Mobile Media OO R1	29	64	6884	1
Mobile Media OO R2	32	74	67301	1
Mobile Media OO R3	38	90	N/A	1
Mobile Media OO R4	40	94	N/A	2
Mobile Media OO R5	48	113	N/A	3
Mobile Media OO R6	54	130	N/A	4
Mobile Media OO R7	65	152	N/A	6
Mobile Media OO R8	81	196	N/A	11
Mobile Media AO R1	29	64	6885	1
Mobile Media AO R2	33	79	N/A	1
Mobile Media AO R3	39	94	N/A	2
Mobile Media AO R4	41	97	N/A	2
Mobile Media AO R5	44	75	N/A	2
Mobile Media AO R6	56	136	N/A	4
Mobile Media AO R7	69	162	N/A	7
Mobile Media AO R8	87	208	N/A	12
VODKA	161	185	N/A	60
Minos 0.0.7	128	967	3288	55
Minos 0.1.0	155	1424	N/A	70
SAT4J	253	2043	N/A	301
Alloy 4	352	8685	N/A	846
Kodkod	476	5595	N/A	1918
Hadoop 0.1	367	3311	N/A	1173
Hadoop 0.2	502	4317	N/A	2481
Hadoop 0.3	533	4734	N/A	2864
Hadoop 0.4	542	4918	N/A	3016
Hadoop 0.5	614	5585	N/A	4394
Hadoop 0.6	672	6236	N/A	5852
Hadoop 0.7	739	7129	N/A	7880
Hadoop 0.8	749	7003	N/A	8228
Hadoop 0.9	790	7400	N/A	9702
Hadoop 0.10	861	8152	N/A	12644
Hadoop 0.11	918	8884	N/A	15459
Hadoop 0.12	1031	10500	N/A	22241

DSM for the AO design of HyperCast, but our algorithm only took one second.

MobileMedia: We evaluated our algorithm over eight releases of a software product line called MobileMedia [33]. MobileMedia contains about 3 KLOC and provides support for handling photo, music and video data on mobile devices, such as cellular phones. Each release evolves from the previous release by adding some new functionalities or by restructuring the previous release to achieve a better modularized structure. Table I reports the running times for dependency analysis on both designs of MobileMedia (one with object-oriented design and one with aspect-oriented design). Our previous work [15] detailed the process of automatically deriving ACNs from UML component diagrams of MobileMedia, which is out of the scope of this paper.

Table I shows that the divide-and-conquer algorithm not only took dramatically more time than our algorithm, but often could not even complete the computation. Their algorithm took almost 2 hours to process release one and almost 19 hours to process release two, despite the fact that only three additional variables were added to the design. On the other hand, our algorithm reduced these running times to just over one second, with a negligible increase in running time. This substantial increase in running time of their algorithm is due to the fact that the number of solutions to a constraint network can be exponential to the number of variables, and emphasizes the need for an algorithm whose performance does not depend on solution count.

After release two, the MobileMedia constraint networks have so many solutions that the divide-and-conquer algorithm can no longer enumerate all solutions with the amount of memory available on our test machine. We further elaborate on this scenario of memory exhaustion below. Since our algorithm does not depend on the number of solutions, we are able to compute the DSMs for all eight releases.

VODKA: We recently had the opportunity to work with a team in the Department of Computer Science at Drexel University, where we manually modeled their senior project using ACNs. The team designed and developed a web-based, service-oriented software system, *VODKA Organizational Device for Keeping Assets* (VODKA) [32], to standardize the financial management method among all student organizations and simplify university auditing. This system contained 154 functional requirements, 11 non-functional requirements, 20 web services, and 13 Java servlets. We modeled the entire system (including requirements, architecture/design, and test procedures) with ACNs by reading their documentation.

As Table I shows, the divide-and-conquer algorithm could not compute the DSM for VODKA: after running for almost two hours, Minos crashed with an out of memory error. We added some debugging code to Minos to report its progress when crashing and found that even with the ACN splitting technique, one of the sub-ACNs contained over 1.1 million solutions. We tried running Minos on a machine with 8GB of memory to see how long it would take to actually derive the DSM for VODKA, but after running for 49 hours and using 7.5GB of memory it was still not finished finding all the solutions. In contrast, without any splitting of ACNs, our algorithm was able to compute the DSM within one minute.

Minos: Minos is a framework for ACN and DSM analysis, developed at Drexel University. It is a refactored version of Cai's Simon [5] tool, written in Java with a flexible plugin architecture to allow for ease of adding new analyses and replacing existing analyses. At two separate occasions, we reverse engineered a UML class diagram from the code base and automatically derived an ACN from it [14].

By examining a breakdown of the running time of Cai and Sullivan's algorithm on the first Minos ACN that we considered (version 0.0.7 with about 10 KLOC), we clearly see the tradeoffs of the divide-and-conquer approach. Cai and Sullivan's algorithm divided Minos into 57 sub-ACNs. Most of these sub-ACNs had few solutions (minimum was 3) and very few sub-ACNs had a large number of solutions (maximum was 23,041), so the total time for finding all the solutions to the sub-ACNs was only about 20 seconds. On the other hand, finding the transitions of the DA took almost half the time (24 minutes) and merging the DAs to took almost the rest of the time (20 minutes). Hence, we see that although using a divide-and-conquer approach reduces the time to solve the constraint network, it introduces additional time to merge the DAs together. Since our algorithm does not take any of these steps, we reduce the running time from 55 minutes to 55 seconds.

Several months later, when Minos version 0.1 was released, we again derived an ACN from the code base and compared the algorithm running times. However, this time Cai and Sullivan's divide and conquer algorithm did not terminate after several days. Accidentally, we forgot to kill the process on our server and stumbled upon it over a month later! After all that time, the process was still running (although the multilevel-queue process scheduler was no longer giving it much CPU time) and using almost all 8GB of RAM on the machine. In contrast, our algorithm took barely more than a single minute to compute the DSM.

Hadoop: The largest system we used to evaluate our approach was Hadoop. Hadoop is an open source map/reduce system for distributed computing, written in the Java programming language. We reverse engineered UML class diagrams from the first twelve releases of Hadoop and automatically derived ACNs from them [14]. Release 0.1 contained 197 classes and interfaces, and release 0.12 contained 601 classes and interfaces.

Due to the number of solutions to the constraint network, the divide-and-conquer algorithm could not produce a DSM on our test machine for most of the releases. Our algorithm, for example, took about 19 minutes to process version 0.1 and 50 minutes to process version 0.2. The time needed to process Hadoop 0.12 took the longest time in our evaluation. As we will discuss in Section VI, by leveraging existing work on computing transitive closure, it is possible to improve the algorithm and further reduce the running time complexity. At this point, we are content with the fact that generating DSMs from a formal model of this size at least becomes possible. We also note that the time for the divide-and-conquer algorithms to report that they failed was longer than the time for our algorithm to complete.

D. Summary

In summary, we used 59 BACNs to compare the performance of our algorithm with an existing algorithm and answered the two evaluation questions posed at the beginning of this section. For all the BACNs, we answer affirmative to the first question that our algorithm computes the correct DSMs, consistent with those produced by Cai and Sullivan's divide-and-conquer algorithm. We also answer affirmatively to the second question that our algorithm outperforms their algorithm for all the studied BACNs. In many cases, the memory requirements to enumerate all solutions of the constraint network prevents the divide-and-conquer algorithm from successfully completing. Since our algorithm's running time (and memory use) does not depend on the number of solutions, our algorithm was able to correctly derive DSMs for all the BACNs, often reducing the running time from hours to seconds from the previous algorithms.

It is worth noting that whether the divide-and-conquer approach will work does not only depend on the number of variables. For example, Cai and Sullivan's algorithm cannot generate the DSM for MobileMedia release 4 with only 40 variables, but can generate a DSM for Minos that has 128 variables. This is due to the fact that Minos is well modularized and its DRs, formalized as dominance relations, decompose the large ACNs into much smaller sub-ACNs.

VI. Discussion

In this section, we discuss possible threats to validity and our future work.

Our results will be more significant if we can show that most ACNs are BACNs. One possible way is to perform the running time comparison using input ACNs randomly generated from a uniform distribution [35], [36]. We chose not to evaluate our algorithm this way but rather to use ACNs in existing literatures or transformed from real software systems because ACNs that the algorithm will encounter in practice may not be uniformly distributed in the problem space. Although an algorithm may have high probability of being efficient for any *randomly* selected input, the inputs it will be given in practice may be exclusively from the subset where it performs poorly.

In other words, although it is possible that the probability of randomly picking an ACN that is also BACN, from all possible constraint networks, could be very low, if by using an ACN, the user just needs to analyze current and future status of each design dimension, it is highly possible that an ACN constructed for this purpose is a BACN. Table I supports this idea that real ACNs are from a non-uniform distribution, and all the ACNs we found had relatively few constraints (compared to the maximum number of possible constraints). These ACNs, modeling multiple heterogenous real software designs, are the inputs for our evaluation.

Despite the benefits of our algorithm over that of Cai and Sullivan's divide-and-conquer algorithm [16], our algorithm only processes BACNs and not all ACNs. On the other hand, their algorithm can work on the general case of ACNs, at the expense of both time and memory. Another benefit of the divide-and-conquer approach is that the decomposed sub-ACNs represent sub-systems, each with all the decisions necessary to implement a function. Decomposing and dependency generation are two orthogonal problems. We can still use the divide-and-conquer method to decompose and investigate sub-systems, and use the BACN algorithm to extract the dependency structure for either sub-systems, or the whole system without the need to solve and recompose the results from sub-systems. The influence graph we used as part of the BACN algorithm generates similar subsets of an ACN. We will further investigate this dimension in the future. Finding more efficient algorithms for the general case of ACNs remains an open future work.

Our future work also includes further optimizing the algorithm and put the algorithm into applications. The current implementation of our algorithm is still in a prototype stage and much optimization can be performed to improve its performance. The relation between variables and constraints in Table I provides insight on one possible way. Since the number of constraints m is much less than the number variables squared $m \ll n^2$, these graphs are sparse so more efficient graph algorithms may improve our algorithm's running time. For example, the most expensive parts of our algorithm are computing the transitive closure for the implication graph and influence graph. Using the Floyd-Warshall algorithm [37], it takes us $O(n^3)$ to compute transitive closures of these graphs, but Johnson's algorithm [37] could offer an improvement to $O(n^2 \log n + nm)$.

Another possible direction for optimization is to use multiple threads to parallelize parts of the algorithm, or avoid the construction of the three graphs. Minos's implementation of Cai and Sullivan's divide-and-conquer algorithm also uses multiple threads as a way of optimization. One area we are exploring is to parallelize the computation of transitive closures [38]. Investigating the benefits of such parallelization is a potential future work.

Besides running time, we are also investigating the reduction of memory requirements of our algorithm. In particular, part of our future work is to determine if all three graphs are required in the algorithm, or if a single structure is sufficient to computing PWDR.

As we mentioned in previous sections, our approach identifies indirect and implicit dependencies that are not revealed by traditional reverse-engineering-based dependency extraction methods. Our hypothesize is that these extra dependencies will improve our ability to predict change impact and communication requirements. Testing this hypothesis is our ongoing work and is out of the scope of this paper, which only focuses on the scalability issue of automatically deriving dependency structures from formal models.

The approach presented in this paper only addresses the scalability issue in the problem of extracting dependency from large-scale formal models without solving the constraints. The remaining problem is: can we conduct *change impact analysis*, as we did before [7], [16], without solving solving the constraint network and generating DAs? We will explore this as our future work.

Although our algorithm shows the possibility of leveraging large-scale formal models in dependency analysis, deriving useful logical dependencies from existing and heterogeneous software artifacts, and automatically transforming and combining them into BACNs remain to be challenging future work.

VII. Related Work

Our work is related to Alloy [12], a lightweight modeling language used to express and enforce constraints among software components. The purpose of Alloy to ensure the correctness and consistency among object states, which is different from our ACN models that aim to model decisions and the assumptions among them. Using Alloy to express decisions and assumptions is possible but more difficult. Alloy also does not provide direct language constructs to express dominance relation or clustering needed by ACN modeling. The original ACN tool, Simon [5], [7], leveraged Alloy as its underlying constraint solver. Minos, the refactored version of Simon, employs Kodkod [34], the same constraint solver that is also used by the latest versions of Alloy, when solving a constraint network is needed.

Unlike many other formal modeling languages, ACNs are designed specifically for analysis of dependencies and software modular structure, based on assumptions among design decisions. For example, while Rosetta [39] also allows the modeling of assumptions and decision domains, it focuses on checking such properties as security, power consumption, and timing constraints. In addition to their differences in terms of purposes, ACNs also differ in that they formalize the notion of design rules from Baldwin and Clark's modularity theory [3], and enable automated DSM derivation, changeability analysis, design modularity and stability measurement [5], [6], [8], [9], [15], [25].

General PWDR algorithms suffer from a *state explosion* problem [22], similar to many model checking algorithms. The ACNs we evaluated often could not be processed by Cai and Sullivan's divide-and-conquer algorithm due to the exceedingly large demand for memory to enumerate all solutions to the constraint network. Approaches such as *binary-decision diagrams* (BDD) [40], [41], and *partial order reductions* [42] offer promise in addressing state explosion in model checking. Research with BDD-based techniques have allowed model checking of systems with as many as 10^{120} states [43]. Applying similar techniques to computing PWDR remains an open future work.

There has been much research on optimization of SAT solvers and constraint satisfiability [34], [44], [45]. While the current Cai and Sullivan algorithm would benefit from such work, there is still an issue of state explosion and storing all solutions. In addition, our experience in using Minos is that with an efficient SAT solver, the time to compute the DA from the set of solutions is often longer than the time to enumerate the solutions. Section V discussed this observation during our evaluation.

VIII. Conclusion

To address the problem that the current approach of generating dependency structure from logical constraint models is difficult to scale due to the needs of constraint solving and solution enumeration, we contributed an algorithm that applies to a restricted but representative form of ACNs, reducing the complexity from NP-complete to polynomial time. We evaluated our approach by generating DSMs from existing ACNs modeling heterogenous software systems, and comparing the DSMs generated and time needed with that of a previous divide-and-conquer algorithm. The results showed that our new algorithm can generate DSMs for much larger models using significantly less time, making it possible to conduct dependency analysis for real software systems.

IX. Acknowledgements

This work was supported in part by the National Science Foundation under grants CCF-0916891 and DUE-0837665.

References

[1] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–8, Dec. 1972.

[2] M. Cataldo, P. Wagstrom, J. D. Herbsleb, and K. M. Carley, "Identification of coordination requirements: Implications for the design of collaboration and awareness tools," in *Proc. ACM Conference on Computer Supported Cooperative Work*, Nov. 2006, pp. 353–362.

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

[4] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Proc. 14th IEEE International Conference on Software Maintenance*, Nov. 1998, pp. 190–197.

[5] Y. Cai and K. J. Sullivan, "Simon: A tool for logical design space modeling and analysis," in *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering*, Nov. 2005, pp. 329–332.

[6] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen, "The structure and value of modularity in software design," in *Proc. 8th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Sep. 2001, pp. 99–108.

[7] Y. Cai, "Modularity in design: Formal modeling and automated analysis," Ph.D. dissertation, University of Virginia, Aug. 2006.

[8] Y. Cai, S. Huynh, and T. Xie, "A framework and tools support for testing modularity of software design," in *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering*, Nov. 2007, pp. 441–4.

[9] K. Sethi, Y. Cai, S. Wong, A. Garcia, and C. Sant'Anna, "From retrospect to prospect: Assessing modularity and stability from software architecture," in *Proc. 8th Working IEEE/IFIP International Conference on Software Architecture*, Sep. 2009.

[10] S. Huynh, Y. Cai, Y. Song, and K. Sullivan, "Automatic modularity conformance checking," in *Proc. 30th International Conference on Software Engineering*, May 2008, pp. 411–420.

[11] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 1999.

[12] D. Jackson, "Alloy: A lightweight object modeling notation," *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 2, pp. 256–290, 2002.

[13] J. M. Spivey, *The Z Notation: A Reference Manual*, 2nd ed. Prentice Hall, 1992.

[14] S. Wong and Y. Cai, "Predicting change impact from logical models," in *Proc. 25th IEEE International Conference on Software Maintenance*, Sep. 2009.

[15] K. Sethi, "From retrospect to prospect: Assessing modularity and stability from software architecture," Master's thesis, Drexel University, Jun. 2009.

[16] Y. Cai and K. J. Sullivan, "Modularity analysis of logical design models," in *Proc. 21st IEEE/ACM International Conference on Automated Software Engineering*, Sep. 2006, pp. 91–102.

[17] A. Mackworth, "Consistency in networks of relations," *Artificial Intelligence*, vol. 8, no. 1, pp. 99–118, 1977.

[18] F. Bacchus and P. van Beek, "On the conversion between non-binary and binary constraint satisfaction problems," in *Proc. 15th AAAI Conference on Artificial Intelligence*, Jul. 1998, pp. 310–318.

[19] S. Even, A. Itai, and A. Shamir, “On the complexity of timetable and multicommodity flow problems,” *SIAM Journal of Computing*, vol. 5, no. 4, pp. 691–703, 1976.

[20] B. Aspvall, M. F. Plass, and R. E. Tarjan, “A linear-time algorithm for testing the truth of certain quantified boolean formulas,” *Information Processing Letters*, vol. 8, no. 3, pp. 121–123, Mar. 1979.

[21] T. Feder, “Network flow and 2-satisfiability,” *Algorithmica*, vol. 11, no. 3, pp. 291–319, 1994.

[22] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Progress on the state explosion problem in model checking,” in *Informatics*, 2001, pp. 176–194.

[23] R. Tarjan, “Depth-first search and linear graph algorithms,” *SIAM Journal of Computing*, vol. 1, no. 2, pp. 146–160, Jun. 1972.

[24] C. V. Lopes and S. K. Bajracharya, “An analysis of modularity in aspect-oriented design,” in *Proc. 4th International Conference on Aspect-Oriented Software Development*, Mar. 2005, pp. 15–26.

[25] K. J. Sullivan, W. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan, “Information hiding interfaces for aspect-oriented design,” in *Proc. 10th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Sep. 2005, pp. 166–175.

[26] A. MacCormack, J. Rusnak, and C. Y. Baldwin, “Exploring the structure of complex software designs: An empirical study of open source and proprietary code,” *Management Science*, vol. 52, no. 7, pp. 1015–1030, Jul. 2006.

[27] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, “Using dependency models to manage complex software architecture,” in *Proc. 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2005, pp. 167–176.

[28] J. Hannemann and G. Kiczales, “Design pattern implementation in Java and AspectJ,” in *Proc. 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Nov. 2002, pp. 161–173.

[29] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Nov. 1994.

[30] K. J. Sullivan, J. B. Dugan, and D. Coppit, “The Galileo fault tree analysis tool,” in *Proc. 29th International Symposium on Fault-Tolerant Computing*, Jun. 1999, pp. 232–5.

[31] J. Liebeherr and T. K. Beam, “Hypercast: A protocol for maintaining multicast group members in a logical hypercube topology,” in *Networked Group Communication*, 1999, pp. 72–89.

[32] Y. Cai and S. Huynh, “A case study of integrating design rule theory and analytical decision models into a software development process,” Drexel University, Tech. Rep. DU-CS-08-02, Apr. 2008, https://www.cs.drexel.edu/content/uploads/Research/DU-CS-08-02.pdf.

[33] E. Figueiredo, N. Cacho, C. Sant’Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. C. Filho, and F. Dantas, “Evolving software product lines with aspects: An empirical study on design stability,” in *Proc. 30th International Conference on Software Engineering*, May 2008, pp. 261–270.

[34] E. Torlak, “A constraint solver for software engineering: Finding models and cores of large relational specifications,” Ph.D. dissertation, Massachusetts Institute of Technology, Feb. 2009.

[35] M. Dyer, A. Frieze, and M. Molloy, “A probabilistic analysis of randomly generated binary constraint,” *Theoretical Computer Science*, vol. 290, no. 3, pp. 1815–1828, 2003.

[36] A. Frieze and M. Molloy, “The satisfiability threshold for randomly generated binary constraint satisfaction problems,” *Random Structures and Algorithms*, vol. 28, no. 3, pp. 323–339, 2006.

[37] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. MIT Press, 2003.

[38] S. Bawa and G. K. Sharma, “A parallel transitive closure computation algorithm for VLSI test generation,” in *Proc. 6th International Conference on Applied Parallel Computing Advanced Scientific Computing*, Jun. 2002, pp. 243–252.

[39] P. Alexander, R. Kamath, and D. Barton, “System specification in Rosetta,” in *Proc. 7th IEEE International Symposium on Engineering of Computer-Based Systems*, Apr. 2000, pp. 299–307.

[40] R. E. Bryant, “Graph-based algorithms for boolean function manipulation,” *IEEE Transactions on Computers*, vol. 35, no. 8, pp. 677–691, 1986.

[41] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, “Symbolic model checking: 10^{20} states and beyond,” in *Proc. 5th Symposium on Logic in Computer Science*, Jun. 1990, pp. 428–439.

[42] E. M. Clarke, O. Grumberg, M. Minea, and D. Peled, “State space reduction using partial ordering techniques,” *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 3, pp. 279–287, 1998.

[43] J. R. Burch, E. M. Clarke, and D. E. Long, “Symbolic model checking with partitioned transition relations,” in *Proc. International Conference on Very Large Scale Integration*, Aug. 1991, pp. 49–58.

[44] D. Marinov, S. Khurshid, S. Bugrara, L. Zhang, and M. Rinard, “Optimizations for compiling declarative models into boolean formulas,” in *Proc. 8th International Conference on Theory and Applications of Satisfiability Testing*, Jun. 2005, pp. 187–202.

[45] S. Khurshid, D. Marinov, I. Shlyakhter, and D. Jackson, “A case for efficient solution enumeration,” in *Proc. 6th International Conference on Theory and Applications of Satisfiability Testing*, May 2003, pp. 272–286.