

Design Rule Hierarchy and Analytical Decision Model Transformation

Sunny Huynh, Yuanfang Cai, and Kanwarpreet Sethi
Dept. of Computer Science
Drexel University
Philadelphia, PA, 19104
{sunny, yfcai, kss33}@cs.drexel.edu

Abstract

We present two contributions to help make design rule theory operable in software engineering practice. First, we develop an approach to automatically transform a design, expressed in the prevailing unified modeling language, into an augmented constraint network (ACN), from which a design structure matrix (DSM) can be generated. Using ACNs formalizes design rule theory and using DSMS enables option-based reasoning. Second, we design an algorithm to derive a design rule hierarchy from an ACN, revealing the impact scope of each design rule and identifying the independent modules. This hierarchy defines an order in which decisions can be made to maximize task parallelism, constrained by the dependency structure of the architecture. We evaluate the accuracy and scalability of our approaches using both small, but canonical, systems and the open-source Apache Ant system.

1. Introduction

Baldwin and Clark's *design rule theory* [4] and the supporting *design structure matrix* (DSM) [4, 13, 31] model have been used to explain how modularity in design creates value in the form of *options*. *Design rules* (DRs) are defined as stable interface decisions. They decouple otherwise coupled decisions, thus creating *independent modules*. Each module supports individual search and replacement with better alternatives. Our prior work presented the *augmented constraint network* (ACN) that formalizes the design rule theory and enables automatic generation of semantically-rigorous DSMs [8, 10]. We refer to both DSM and ACN as *analytical decision models*.

Researchers showed that the theory and the models are useful in software design analysis. Sullivan et al. [32], Lopes et al. [22], and Cai et al. [9] used the theory and the models to precisely capture Parnas's information hiding criterion and to quantitatively compare design alternatives.

Sangal et al. [28], MacCormack et al. [23], and LaMantia et al. [20] used source code reverse-engineered DSMs to reveal and compare large-scale software modular structure, and to explain software evolution phenomena and modularization activities.

However, there are several obstacles that hinder software designers from exploiting the power of design rule theory and analytical decision models. First, manually constructing design-level DSMs has been shown to be error-prone and subject to ambiguity [8]. In addition, ACN modeling requires identifying and expressing logical relations among design decisions. For software engineers, who are trained to use prevailing models such as UML, constructing these analytical decision models and applying the design rule theory present a steep learning curve.

Second, identifying independent modules is not trivial in large-scale systems. Consistent with Parnas's definition [26], Baldwin and Clark [4] define a module as an independent task assignment, which consists of a set of decisions that can be made together; decisions in separate modules are made in parallel with each other. However, this definition is not always consistent with the concept of module that software engineers use; these modules often stand for classes, aspects, or components that are not always independent. The difficulty of identifying independent modules is directly related to the difficulty of differentiating design rules in terms of their impact scope. According to its definition, all classes with a public interface are providing a design rule. However, the impact scope of an interface varies dramatically. For example, an abstract class with many descendants can be more critical than an interface that is only used by a few classes.

We present an approach to address each of these problems. First, we develop an approach to automatically transform a UML class diagram into an ACN. Second, we design an algorithm to identify independent modules from an ACN, and to cluster the design rules into a *design rule hierarchy* according to their impact scopes, which can be visualized through a DSM. We evaluate our UML transformation

and DR hierarchy algorithm using both small systems and the open-source Apache Ant [3].

Our design rule hierarchy is different from other well-known hierarchical structures, such as the “uses” hierarchy defined by Parnas [27] and hierarchies defined according to conceptual domains [7, 11]. For example, if a decision *A* in a GUI layer is the only decision to depend on a decision *B* in the business layer, then our DR hierarchy algorithm will aggregate *A* and *B* into a single independent module because these decisions can and should be made and changed together. Within each layer of our hierarchy, modules partition the decisions, such that each module can be decided in parallel with other modules within the layer. Thus, this hierarchy shows how to assign tasks for maximum work to be done in parallel.

The rest of this paper is organized as follows. Section 2 gives an overview of our modularity analysis framework and Section 3 illustrates our approach with a small example. Section 4 describes the formalization of our UML transformation and DR hierarchy clustering algorithm. Section 5 presents our experiments on KWIC and Apache Ant. Section 6 discusses our results and Section 7 addresses related work. Section 8 concludes.

2. Framework

Figure 1 shows our overall modularity analysis framework and how the approaches we present in this paper fit into it. The framework takes a UML model as input and generates a DSM clustered with our design rule hierarchy. *ACN Decomposer* is an algorithm that takes an ACN and decomposes it into a set of sub-ACNs according to formalized design rules, as presented by Cai et al. [10]. Cai et al. [10] also presented an algorithm, represented as *Dependency Analyzer* in the figure, to derive a pairwise dependence relation from ACNs. The *DSM Generator* generates DSMs from the pairwise dependence relation and a selected clustering method. The DSM model can then be used to conduct a number of modularity and evolution analyses. We use the implementation of these algorithms from the Simon [8, 10] tool.

Our contribution in this paper are the two highlighted processes. *UML to ACN* is a program that converts a UML model into an ACN model. *Design Rule Hierarchy* is an algorithm that takes two inputs: (1) a set of sub-ACNs decomposed by *ACN Decomposer* and (2) the pairwise dependence relation derived from the ACN, and outputs a DSM clustered by the DR hierarchy.

3. Overview Through an Example

Figure 2 shows a UML class diagram of a small system for building a maze in a computer game; this system

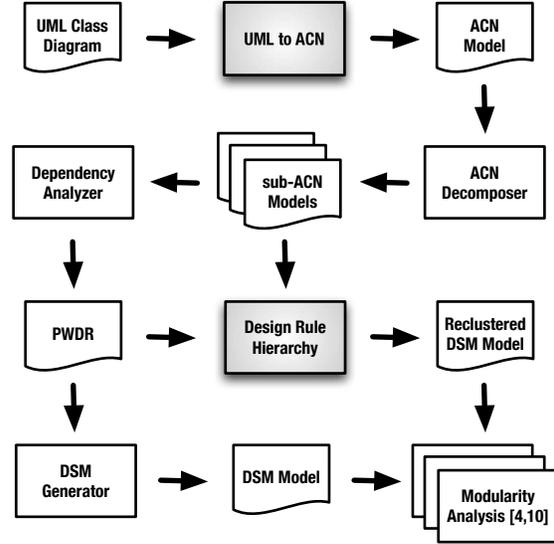


Figure 1. Modularity Analysis Framework

uses the abstract factory pattern, as described in Gamma et al. [15]. A maze is defined as a set of rooms; a room knows its neighbors, such as another room, a wall, or a door to another room. The class `MapSite` is the common abstract class for all the components of the maze. The UML class diagram shows two variations of the maze game supported by the abstract factory pattern: an enchanted maze game (`EnchantedMazeFactory`) with a door that can only be opened and locked with a spell (`DoorNeedingSpell`), and a room that has a magic key (`EnchantedRoom`); and a bombed maze game (`BombedMazeFactory`) that contains a room with bomb set in it (`RoomWithABomb`) and a wall that can be damaged if a bomb goes off (`BombedWall`).

UML modeling is not designed to aid independent task assignment. For example, if a team member is assigned the task of implementing the enchanted maze game, he/she has to examine the diagram to determine all the classes that have to be designed. In addition, he/she must be aware of other classes, such as `MapSite`, that the enchanted maze game components must interact with. These classes may be designed by other team members, creating dependencies between tasks. UML models can scale to where tracing and understanding the relations among the classes, in order to determine these dependencies, can become difficult [12].

The following subsections describe design rule theory, analytical decision models, and the DR hierarchy. We describe how a UML class diagram can be converted to an ACN and a DSM, and how to derive a DR hierarchy.

3.1. Decisions Models and DR Hierarchy

Design Structure Matrix Figure 3(a) shows a design structure matrix (DSM) automatically generated from the

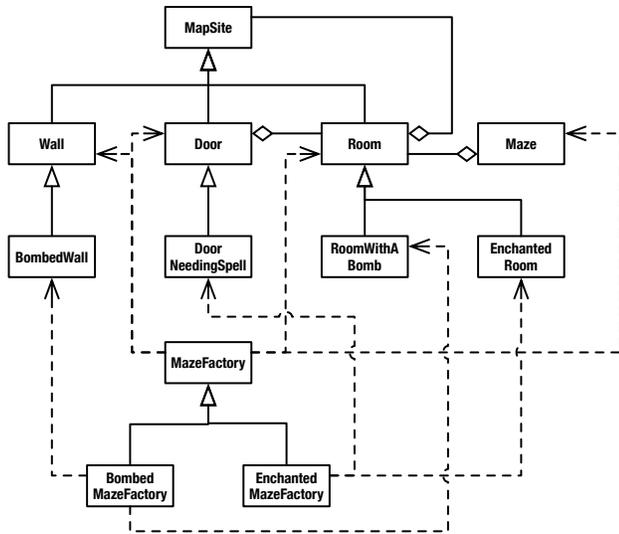


Figure 2. Maze Game UML Class Diagram

UML diagram in Figure 2. A DSM is a square matrix in which rows and columns are labeled with design dimensions where decisions must be made, and a marked cell models that the decision on the row depends on the column. In this DSM, each class is modeled using two design dimensions: an interface and an implementation. For example, the cell in row 11, column 2 indicates that the implementation of the `Room` class (`Room_impl`) depends on the interface of the `MapSite` class (`MapSite_interface`).

Blocks along the diagonal represent aggregated design decisions. Figure 3(a) shows two levels of such aggregation. Each block at the inner level represents a module, which is an independent task assignment. At the outer level, each block represents a layer in our DR hierarchy. We describe the significance of these layers in the following paragraphs.

Design Rule Design rules are captured by a DSM using asymmetric dependencies that decouple modules. For example, the `Room_impl` variable influences both `BombedRoom_impl` and `EnchantedRoom_impl`, but is not influenced by them. Therefore, once the common room characteristics are implemented by the parent `Room` class, the `BombedRoom_impl` and `EnchantedRoom_impl` only need to implement their own special features; they need not know the existence of each other. As a result, the `Room_impl` serves as a design rule that decouple the implementations of `EnchantedRoom` and `BombedRoom`.

Design Rule Hierarchy The DSM shown in Figure 3(a) is clustered into a 4-layer DR hierarchy. The four outer groupings in Figure 3(a) show the *layers* in which tasks can be completed in parallel; the clusters within each layer only depend on the decisions within the layers to the left of it in the DSM. The tasks within each layer of the DSM can

be completed in parallel because there are no inter-module dependencies within each layer.

The first layer identifies the design rules that are most influential and should remain stable. In Figure 3(a), the first layer consists of the variables `Maze_interface` and `MapSite_interface`. Changing these design rules can have drastic effects on a system. For example, changing a public method signature in the `MapSite` class may require changes to almost all parts of the software (as shown by the numerous marks under column 2).

The second layer, from row 3 to row 6, contains the decisions that only depend on the top layer decisions and can be made in parallel. Each cluster within the layer contain the decisions that should be made together. For example, the DSM shows that the `MazeFactory_interface` and `MazeFactory_impl` decisions should be made together. Although `MazeFactory` and `DoorNeedingSpell_interface` do not belong to the same layer of an inheritance hierarchy, they are in the same DR hierarchy layer because once the DRs in the previous layer are determined, these decisions can be made at in parallel.

The last layer of the hierarchy identifies the independent modules, consistent with Parnas's definition. Not only can these modules be designed and developed independently, but they can also be swapped out for different implementations without affecting the rest of the system. Although `BombedWall_impl` and `Wall_impl` belong to different layers of the inheritance hierarchy, they are clustered into the same module. Even though `Wall_impl` is a parent class, it does not decouple multiple modules, and is only used by the `BombedWall_impl`. As a result, `Wall_impl` is not a design rule in the current system, and the engineers of these two classes can work together for a better `Wall` implementation without worrying about unwanted side effects.

Hence, the DR hierarchy-clustered DSM identifies independent modules and reveals how to schedule tasks for maximum work to be done in parallel. Manually generating, marking and clustering a DSM, even at this size, is not practical. The DSM shown in Figure 3(a) is generated from an augmented constraint network translated from the UML class diagram, and automatically clustered using our DR hierarchy algorithm. Next, we describe the augmented constraint network.

Augmented Constraint Network The augmented constraint network (ACN), developed by Cai et al. [8, 10], formalizes the concept of design rules and enables automatic DSM derivation. Figure 3(b) shows part of the ACN derived from the UML diagram show in Figure 2 and Figure 3(a) shows the corresponding derived DSM. An ACN consists of a constraint network that models design decisions and their relations, a dominance relation that formalizes the concept of design rule, and a cluster set in which each cluster represents a different way to partition a design.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Maze_interface	1	.																						
MapSite_interface	2		.																					
Wall_interface	3			x	.																			
Door_interface	4			x		.																		
Room_interface	5			x			.																	
MapSite_impl	6			x				.																
MazeFactory_interface	7								.															
MazeFactory_impl	8	x	x	x	x	x			x	.														
BombedWall_interface	9			x	x						.													
EnchantedRoom_interface	10			x								.												
Room_impl	11			x			x	x					.											
DoorNeedingSpell_interface	12			x		x								.										
RoomWithABomb_interface	13			x											.									
BombedWall_impl	14			x	x				x						.	x								
Wall_impl	15			x	x										.									
EnchantedRoom_impl	16			x		x	x									.								
EnchantedMazeFactory_impl	17	x	x	x	x	x		x	x							.	x							
EnchantedMazeFactory_interface	18									x							.							
RoomWithABomb_impl	19			x			x	x									.							
BombedMazeFactory_interface	20									x								.						
BombedMazeFactory_impl	21	x	x	x	x	x		x	x	x								.						
Maze_impl	22	x	x			x													.					
DoorNeedingSpell_impl	23			x		x	x	x												.				
Door_impl	24			x		x	x	x													.			

(a) DSM

```

1. MapSite_interface : {orig, other};
2. MapSite_impl : {orig, other};
3. Room_interface : {orig, other};
4. Room_impl : {orig, other};
5. Maze_interface : {orig, other};
6. Maze_impl : {orig, other};
7. Room_impl = orig =>
  MapSite_interface = orig;
8. Room_impl = orig =>
  MapSite_impl = orig;
9. Maze_impl = orig =>
  Room_interface = orig;
10. (MapSite_impl, MapSite_interface);
11. (Room_impl, Room_interface);
12. (Room_interface, MapSite_interface);

```

(b) Partial ACN

Figure 3. Maze Game Analytical Decision Models

A constraint network consists of a set of design variables, which model design dimensions or relevant environment conditions, and their domains; and a set of logical constraints, which model the relations among variables. In Figure 3(b), lines 1–6 are the partial maze game variables, and lines 7–9 are some sample constraints. For example, line 9 models that the implementation of the `Maze` class assumes that the interface of the `Room` class is as originally agreed.

We augment the constraint network with a binary *dominance relation* to model asymmetric dependence relations among decisions, as shown in lines 10–12. For example, line 11 indicates that the decision for how to implement the `Room` class cannot influence the design of its interface; in other words, we cannot arbitrarily change the `Room` class’s public interface to simplify the class’s implementation because other components may rely on that interface.

From the constraint network and the dominance relation, we can formally define *pairwise dependence relation* (PWDR): if $(x, y) \in PWDR$ then y must be changed in some minimal restoration of consistency to the constraint network which was broken by a change in x . We have shown that a DSM can be automatically derived from an ACN [8, 10] where the matrix is populated by the PWDR and the columns and rows are ordered by a selected cluster from the cluster set.

3.2. Converting UML to ACN

The basic idea in converting UML to ACN is to formalize each UML relation, such as generalization and association, using a constraint relation and dominance relation. We use the generalization relation between the `Mapsite` and

`Room` classes as an example.

To translate a UML relation into a constraint network, we first transform a class into two variables; the interface variable ends with `_interface` and the implementation variable ends with `_impl`. Then, we model the relations among classes and interfaces as logical constraints. For example, the `Room` class inherits from the `MapSite` class so a change in either the interface or implementation of `MapSite` will influence the implementation of `Room`. In other words, the implementation of `Room` makes assumptions about the interface and implementation of `MapSite`. Lines 7–8 in Figure 3(b) are the logical expressions translated from these relations. Although a change in the implementation of a parent class implicitly propagates to its children and a child’s code may not need to be changed, the designer should be aware of this implicit change to avoid unexpected side effects.

To generate the dominance relation, we dictate that interfaces dominate implementations. We also decide that the implementation decision of a parent class dominate the implementation decision of a child because the parent may have other children. Forcing a parent class to change due to a change in a child may cause unwanted side effects in other children.

The ACN model generated from the maze game UML class diagram consists of 24 variables, 50 logical expressions, and 49 pairs of dominance relations. Figure 3(a) shows the DSM generated from this ACN, and clustered into a DR hierarchy.

3.3. DR Hierarchy Clustering

Our motivation in generating a DR hierarchy stems from the need to identify independent task assignments within a design. We first need to identify all decisions needed for each task. Then we identify which of these decisions are shared by other tasks and which can be made independently and concurrently. The first step is done by our prior work of decomposing an ACN into a set of sub-ACNs [10]. We present an algorithm to address the second issue and build on the ACN decomposition approach.

Identify Decisions Needed by a Task Cai et al. [10] introduced an algorithm to automatically decompose an ACN model into sub-ACNs; each sub-ACN was shown to contain a set of decisions needed to accomplish a particular task. We refer to this algorithm as the Decompose-Modules algorithm. The basic idea is to model the constraint network part of an ACN using a directed graph. In this graph, each vertex represents a design variable. Two variables are connected if and only if they appear in the same constraint expression. Then the edges of the directed graph are removed using the dominance relation of the ACN: if A cannot influence B , then the edge from A to B is removed from the graph. We then compute the condensation graph of this graph. Figure 4(a) shows a partial maze game condensation graph generated from the maze game ACN. Note that the edge directions in the graph may seem counter-intuitive. This is because the edges do not represent the direction of dependence but rather the direction of possible influence. In other words, if an edge (u, v) exists in the graph then a change in the decision for variable u may potentially influence the decision on variable v .

To generate sub-ACNs, we put all the variables along the paths ending with the same minimal elements into a sub-ACN with the relevant subset of constraints, dominance relation and cluster set. As a result, the ACN is decomposed into a set of sub-ACNs that can be solved individually. We observe that each minimal element of the condensation graph represents a feature, and all the chains ending with a minimal element contain all the decisions needed to realize the feature. For example, Figure 4(a) shows that one of the sub-ACNs will contain the variables `BombedWall.impl`, `BombedWall.interface`, `Wall.interface`, and `MapSite.interface`. This sub-ACN contains all the decisions needed to implement the `BombedWall.impl` feature.

Simply identifying all the decisions needed for a feature does not guarantee that the task can be implemented or changed independently because some of the decisions may be shared by other tasks. For example, the `BombedWall.impl` sub-ACN contains decisions, such as `MapSite.interface`, that cannot be made or changed independently because are shared by other tasks, as shown by

the overlapping variables in the condensation graph. We differentiate the sub-ACNs in Figure 4(a) with different line styles and label them as s_1, s_2, s_3 for illustration.

Identify Shared Decisions Our goal is to identify a hierarchy from the condensation graph, which is generated as a by-product of the Decompose-Modules algorithm, and further decompose these sub-ACNs into independent tasks. We call this hierarchy the design rule hierarchy because the hierarchy is determined by the design rules, formalized as the dominance relation of the ACN.

Intuitively, our algorithm identifies each region of intersection in the condensation graph and separates each into an individual group. For example, there are two regions of intersection in Figure 4(a). In the intersection $s_1 \cap s_2 \cap s_3$, `MapSite.interface` is shared by all three sub-ACNs so we create a new module exclusively consisting of it. Since the variable in this intersection is shared by all the sub-ACNs, its development must be completed first. The other region of intersection, between s_2 and s_3 , contains `Room.interface` so we create a new module consisting of it. Similarly as before, since s_2 and s_3 share this variable, it must be designed before the design of the concrete `Room` classes. Since s_2 and s_3 no longer share any variables, they are identified as independent tasks that can be designed in parallel. Figure 4(b) shows the hierarchy our algorithm produces from Figure 4(a) after aggregating these intersections into separate modules.

Since the resulting graph after applying this algorithm is directed acyclic, we can apply a modified breadth first search of the vertices and get a partial ordering. In other words, if we complete the tasks in Figure 4(b) from the top down, then we will not have any issues with waiting for dependencies to be completed. Figure 3(a) shows the DSM of the full maze game example in which the variables are clustered according to the result of our algorithm. All the dependencies in the DSM are either below the diagonal or within the inner groups. Sangal et al. [28] refer to this as being in *block triangular* form. As we will prove in Section 4, our algorithm always produces a clustering that puts a DSM in block triangular form.

4. Formalization

In this section, we formalize of UML-to-ACN transformation and the design rule hierarchy clustering algorithm.

4.1. UML Transformation

For the sake of space, we only present the ACN formalization of UML *generalization* relation. Our previous work [17] details the formalization of additional UML relations. Table 1 shows a *generalization* relation depicted in

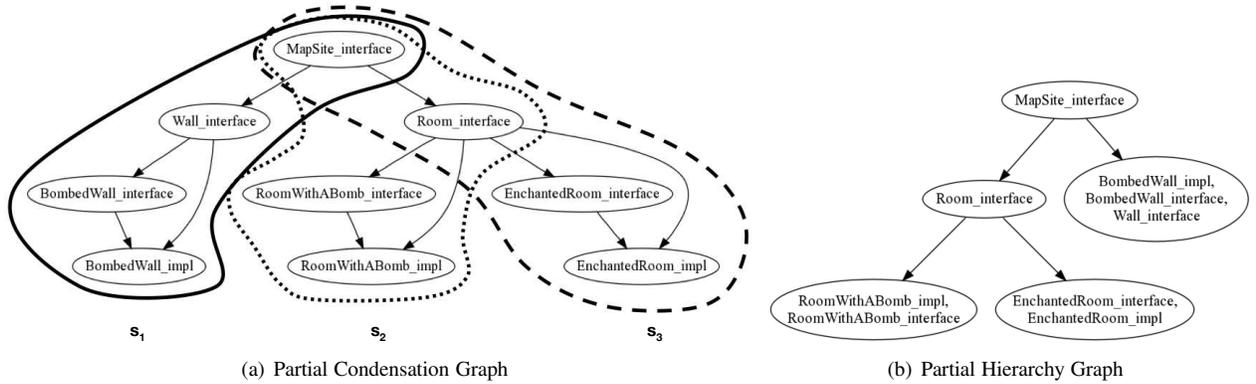


Figure 4. Maze Game Decomposition Graphs

UML, in which **A** is the general element and **B** is the specific element.

UML	Augmented Constraint Network
	<p>Constraint Network:</p> <pre> B_impl = orig => A_interface = orig B_interface = orig => A_interface = orig B_impl = orig => A_impl = orig </pre> <p>Dominance Relation:</p> <pre> (B_impl, A_interface) (B_impl, A_impl) (B_interface, A_interface) </pre>

Table 1. Generalization

Since **B** inherits from **A**, the decision on **A**'s interface dominates and influences both **B**'s interface and implementation, as modeled using the first two logical constraints and first two dominance pairs shown in the table. The implementation decision of **A** also influences the implementation of **B** as expressed in the third constraint. **A_impl** also dominates **B_impl** because a change in **B**'s implementation should not force changes in **A**'s implementation, which may be inherited by other classes. The dominance relation is constructed accordingly.

4.2. Design Rule Hierarchy Clustering

In this subsection we formally define the *design rule hierarchy*, present the DR hierarchy algorithm, prove its correctness, and analyze its complexity. A DR hierarchy is a directed acyclic graph (DAG) where each vertex models a *task*; each task is defined as a set of design decisions that should be made together. Edges in the graph model an “assumes” relation: an edge (u, v) models that the decision v *assumes* decision u . Based on ACN modeling, a change in

the choice for u may cause a change in the choice for v . The layers within the DR hierarchy obey the following rules:

- Layer 0 is the set of tasks that *assume* no other decisions.
- Layer i ($i \geq 1$) is the set of all tasks that *assume* at least one decision in level $i - 1$ and *assume* no decisions at a layer higher than $i - 1$. Within any layer, no task assumes any decisions within another task of the same layer. Hence, the tasks within the same layer can be designed independently and in parallel.
- The highest layer is the set of independent modules. No decisions outside of these modules makes assumption about any decisions within these modules.

4.2.1. DR Hierarchy Algorithm

Our DR hierarchy algorithm starts by identifying all the decisions needed for a task using the Decompose-Modules algorithm created by Cai and Sullivan [8, 10] to decompose a ACN into sub-ACNs. Decompose-Modules takes as input a directed graph G (representing the constraint network) and the dominance relation pairs. It creates from G a condensation graph C , and outputs a set of sub-ACNs S . Our algorithm takes both C and S as input and outputs a clustering that conforms to the formal definition of DR hierarchy. Figure 5 contains the pseudocode of our algorithm.

As stated in Section 3, intuitively, our algorithm separates each region of intersection of the sub-ACNs into a separate cluster. To identify which region a vertex of C belongs to, we assign to each vertex an identifying bit-field of $|S|$ bits (represented by the integer *region* in the pseudocode). For each sub-ACN $s_i \in S$, if a vertex is contained in that sub-ACN then the i -th bit of its bit-field will be set to 1. For example, in Figure 4(a), `Room_interface` is contained in sub-ACNs s_2 and s_3 so it would have an identify-

```

DECOMPOSE-HIER( $C, S$ )
1  create hash table  $T$ 
2  for each  $v \in V[C]$ 
3      do  $region \leftarrow 0$ 
4          for each  $s \in S$ 
5              do  $region \leftarrow region \times 2$ 
6                  if  $v \in s$ 
7                      then  $region \leftarrow region + 1$ 
8               $T[region] \leftarrow T[region] \cup \{v\}$ 
9  create graph  $H$ 
10  $V[H] \leftarrow$  items in  $T$ 
11 for each  $(u, v) \in V[H] \times V[H]$ 
12     do if  $\{(i, j) \in E[C] \mid i \in u \wedge j \in v\} \neq \emptyset$ 
13         then  $E[H] \leftarrow E[H] \cup \{(u, v)\}$ 
14 return  $H$ 

```

Figure 5. Algorithm Pseudocode

ing bit-field of 110 and `wall_interface` is only contained in sub-ACN s_1 so it would have a bit-field of 001.

After identifying regions, we build a new graph H in which each vertex represents a region. The final for-loop in the pseudocode populates the edges of H based on edges in the condensation graph C . The graph H contains the hierarchical structure of tasks based on the “assumes” relation. To derive the DR hierarchy clustering from H , we first isolate the independent modules, then perform a modified breadth-first search (BFS) on the graph. The modification is that a vertex is not explored until all its incoming neighbors have been explored. By using BFS we can explicitly identify the layer that each element belongs to.

4.2.2. Proof of Correctness

To show that our approach correctly finds a hierarchical structure of a software design, we prove Theorem 1 by contradiction. To simplify this proof, we first prove Lemma 1.

Lemma 1. *If v_j, \dots, v_k is a path in the condensation graph C , then for any sub-ACN $s \in S$ if $v_k \in s$ then $v_j \in s$.*

Proof of Lemma 1. Let u be a minimal element in C such that there is a path $v_k \rightsquigarrow u$ (without loss of generality, assume that a path can consist of a single vertex if $v_k = u$). There must be at least one unique $u \in C$ because C is a DAG. The Decompose-Modules [8, 10] algorithm builds a sub-ACN from u by putting all vertices that are connected to u in the sub-ACN. Since v_k is connected to u , it is in the sub-ACN; since v_j is connected to v_k and v_k is connected to u , v_j is also in the sub-ACN. \square

Theorem 1. *The hierarchy graph H is a DAG.*

Proof of Theorem 1. Since the input condensation graph C does not contain any cycles, the only way that a cycle can be formed in H is by the clustering of vertices of C . For example, if a simple path $p = v_1, v_2, \dots, v_k$ exists in C , and a vertex is created in H containing v_1 and v_k then a cycle would be formed. We assume by contradiction, that v_1 and v_k are clustered together in H . Then by definition that for all input sub-ACNs $s \in S$, $v_1 \in s$ iff $v_k \in s$. For a cycle to be formed, at least one vertex in v_2, \dots, v_{k-1} must not be clustered with v_1 and v_k ; let v_i be this vertex. If v_i is not clustered with v_k then there exists at least one sub-ACN $s' \in S$ such that one, but not both, of v_i and v_k is in s' . We consider each case separately.

- $v_k \in s' \wedge v_i \notin s'$
Since v_i is in our path p , there exists a path $v_i \rightsquigarrow v_k$. By Lemma 1, if $v_k \in s'$ then v_i must also be in s' . Hence, this scenario never occurs.
- $v_k \notin s' \wedge v_i \in s'$
Since v_i is in our path p , there exists a path $v_1 \rightsquigarrow v_i$. By Lemma 1, if $v_i \in s'$ then v_1 must also be in s' but this contradicts our original assumption. The contradiction occurs because we assumed that for all sub-ACNs $s \in S$, $v_1 \in s$ iff $v_k \in s$, but this scenario would have $v_k \notin s'$ but $v_1 \in s'$.

Therefore, v_1 and v_k cannot be clustered together to form a cycle in H . This proof can easily be extended to show that cycles cannot be formed by clustering together ends of multiple paths. For sake of space, we do not present that here. Since the graph is a DAG, we guarantee that the corresponding DSM will be clustered into block triangular form. \square

4.2.3. Complexity Analysis

To show the running time for our algorithm we first bound the size of its inputs. All $|V[C]|$, $|S|$, and $|V[H]|$ are bounded by the number of variables in the ACN $|V|$ because each vertex or sub-ACN must contain at least one variable. From this, we know that each of the first two for-loops of our algorithm will run in $\Theta(|V|)$ times and the last for-loop runs in $\Theta(|V|^2)$ time. Breadth-first search runs in linear time so the total running time of our algorithm is $\Theta(|V|^2)$.

5. Preliminary Evaluation

In this section, we present our preliminary evaluation strategy and results for both the UML transformation approach and the design rule hierarchy algorithm.

5.1. UML Transformation

The purpose of our UML transformation evaluation is to assess (1) if the translated ACN/DSM model can faithfully

capture the dependence relations among classes determined by the UML diagram; and (2) if the approach can scale to the size of a real project.

5.1.1. Accuracy Assessment

Our strategy to achieve the first goal is to compare the DSM that is derived from the UML class diagram with the DSM that is reverse engineered from source code. We use the Lattix [21] tool to derive the source code DSM. This evaluation is under the premise that the implementation confirms to the design. Throughout this section, we refer to the UML-translated DSM as the *design DSM* and the reverse engineered DSM as the *source DSM*. We answer the following evaluation questions through the comparison: (1) can the design DSM pick up all the dependencies picked up by the source DSM? (2) what causes the discrepancy between them, if any?

We select a small, but canonical, system to answer these questions. A small system is used so that we can manually analyze any DSM differences. A canonical system is used to ensure the correctness of the UML diagram and implementation. The evaluation is valid as long as all the UML relations are covered. The subject we select is the maze game used in design pattern book of Gamma et al. [15]. The authors of the book provided an implementation so we are assured that it confirms to the UML diagram.

Figure 6 shows the merged design and source DSMs for the maze game to highlight their differences. To make the two DSMs comparable, we collapse each pair of interface and implementation variables of each class into a single variable in the design DSM. After this clustering, the two DSMs have the same set of variables so that we can overlap them and show the differences. The figure shows that the design DSM has a superset of dependencies of the source DSM. The dependencies with a dark background are those which the design DSM produced but the source DSM did not. The differences fell into two categories:

1. The dependencies with a dark background and an “x” mark are due to our approach’s ability to explicitly reveal *indirect* dependencies. For example, the design DSM shows a dependency from `EnchantedRoom` to `MapSite`. Although `EnchantedRoom` derives from `Room`, rather than `MapSite` directly, changes to `MapSite` could require a change to both `Room` and `EnchantedRoom`. For example, if a public method signature is changed in `MapSite` and `EnchantedRoom` overrides this method, then its method would also need to be updated.
2. The dependencies indicated by a dark background and an “o” mark are due to our approach’s ability to explicitly reveal *implicit* dependencies. For example, our design DSM shows that `BombedMazeFactory` depends on `Door` because although `BombedMazeFactory`

may not have explicitly overridden the method to create a `Door` from `MazeFactory`, there was an implicit assumption that `Door` stay as originally agreed. `BombedMazeFactory` may not have overridden the `Door` creation because it assumed that all doors could be opened. But if the default behavior of opening a `Door` is changed, then the `BombedMazeFactory` may need to change. We explicitly show this dependency in order to prevent unexpected side effects as such.

We can conclude from the experiment that the UML-translated design DSM faithfully picks up the dependencies determined by the UML class diagram, including both implicit and indirect dependencies.

	1	2	3	4	5	6	7	8	9	10	11	12
MapSite	1	.										
Room	2	x	.									
Door	3	x	x	.								
Maze	4	o	x		.							
Wall	5	x				.						
BombedWall	6	x				x	.					
DoorNeedingSpell	7	x	x	x				.				
EnchantedRoom	8	x	x						.			
MazeFactory	9	o	x	x	x	x				.		
RoomWithABomb	10	x	x								.	
BombedMazeFactory	11	o	x	o	o	x	x			x	x	.
EnchantedMazeFactory	12	o	x	x	o	o		x	x	x		.

Figure 6. Maze Game Clustered DSM

5.1.2. Scalability Assessment

To evaluate the scalability of the approach, our strategy is to apply it to a large-scale UML model. Due to the difficulty of finding a publicly available UML model for a real project, we reverse engineered a UML class diagram from the open-source Apache Ant system [3], version 1.7.0. Our previous work detailed the reverse engineering process [17]. The Apache Ant UML model contains 626 classes and interfaces, and over 3000 inter-component relations. We transform it into an ACN with 1200 variables and 4400 constraints. Figure 7 shows part of the Apache DSM we derived from the ACN. The full DSM is too large to fit in the paper, but can be viewed at the first author’s website¹. Our tool took 15 minutes to perform the transformation, showing the feasibility of the approach for real projects.

5.2. Design Rule Hierarchy Clustering

Our evaluation goal in assessing the DR hierarchy algorithm is to determine (1) if the DR hierarchy algorithm can identify independent modules and how the design tasks can be partitioned to maximize parallel work; and (2) if the approach can scale to the size of a real project.

¹http://rise.cs.drexel.edu/~sunny/papers/icse2009_ant.xlsx

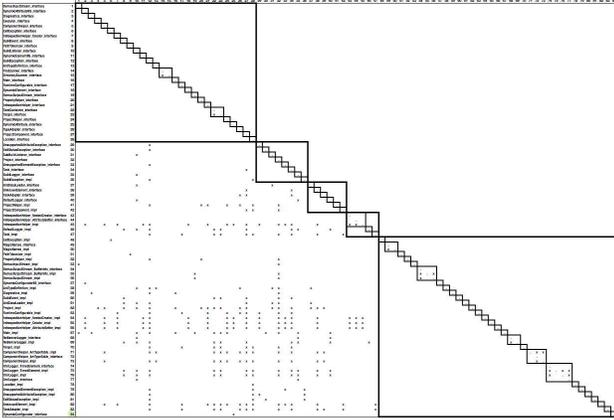


Figure 7. Partial Apache Ant DSM

5.2.1. Accuracy Assessment

Our strategy is to compare the computed DR hierarchy clustered DSM with a previously validated DSM model. Given an existing DSM in which the design rules are manually identified and the modules are manually clustered, we answer the following evaluation questions through the comparison: (1) does the DR hierarchy-clustered DSM identify the same set of DRs and independent modules? and (2) if not, what causes the discrepancy between them?

We choose the DSM models constructed by Sullivan et al. [8, 10, 32] that model Parnas’s canonical and extensively-studied *keyword in context* (KWIC) [26] system. In the DSMs presented in previous works, all the environment variables were aggregated into a single module. To ease the comparison, we slightly modify the automatically generated hierarchy by moving all the environment variables to a standalone module. Figure 8 shows a DSM of KWIC with this modified hierarchy. Since environmental conditions do not depend on design decisions, the DSM is still in block triangular form.

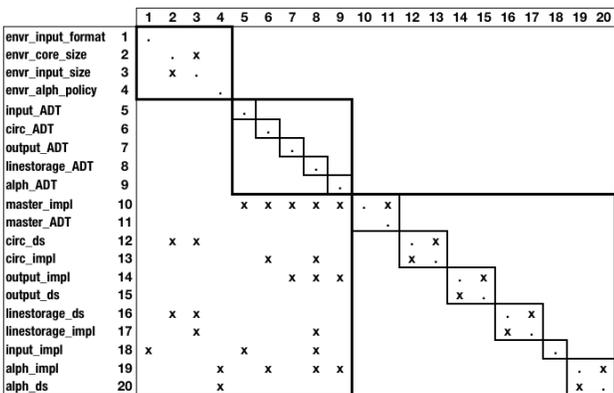


Figure 8. KWIC Design Structure Matrix

In comparing our DR hierarchy-clustered DSM with the manually clustered DSM published in previous works [8, 10, 32], we notice only two differences.

1. In previous DSMs, all the design rules were grouped into a single module whereas our approach generates five separate modules that are design rules. The separated clusters in our model indicate how design tasks done in parallel. For example, we can have someone start development on the `input_impl` module after the `linestorage_ADT` and `input_ADT` design rules are determined rather than waiting for the other design rules to be determined also.
2. Our approach does not identify `master_ADT` as a design rule as the previously published DSMs did. We observe that the only dependent of `master_ADT`, in the specified design of KWIC, is `master_impl`. Based on the definition of design rules, our algorithm’s classification of `master_ADT` is correct: design rules are meant to decouple subordinate modules but `master_ADT` does not decouple two modules. As a result, our approach correctly classifies it as not being a design rule.

In summary, we can conclude that our approach accurately identifies the design rules and independent modules.

5.2.2. Scalability Assessment

As preliminary evaluation for the scalability of our approach, we apply it to the Apache Ant ACN we derived in our UML transformation evaluation. The input ACN consists of 1200 variables and 4400 constraints. Figure 7 shows the part of the system’s DSM as clustered with our design rule hierarchy. After applying our hierarchy clustering, the DSM is in block triangular form. The current implementation of our DR hierarchy algorithm took less than 15 seconds to derive this hierarchy, showing the feasibility of the approach for use in real projects.

As an initial evaluation on the usefulness of our approach, we examined the derived DR hierarchy for Apache Ant to see if the clustering provides insights into the quality of the system’s design. Despite having over 600 classes and 1200 variables, the hierarchy consists of only nine layers, indicating much of the system could have been developed in parallel. In addition, our hierarchy contained 837 tasks with 436 of them being independent modules, containing 671 variables. Since more than half of the variables in the ACN are in independent modules, this suggests that the system’s design is very well modularized to allow for evolution and maintenance. Together, the low depth of the hierarchy and the high number of independent modules suggest that the system has well defined interfaces (design rules) that decouple the other modules in the system. Although these properties are interesting indications of the actual quality

of the design, we have yet to confirm these properties with empirical metrics.

6. Discussion and Future Work

In this paper, we only considered UML class diagrams, but there are many UML diagrams, such as sequence diagrams and component diagrams. Automatically transforming other UML diagrams into ACNs is part of our future work. One of our concurrent work [29] investigates the transformation of UML component diagrams into ACNs and defines metrics for assessing software architecture stability based design rule hierarchy clustering.

Since a UML class diagram only models part of a design space, and there many implicit decisions and constraints a UML class diagram does not capture, the resulting ACNs thus can be insufficient. This paper does not evaluate how effective it is to analyze design modularity based on UML model only. On the other hand, the UML-generated ACN model can always be extended and complemented with new variables and constraints, as shown in our recent work [29], and the UML based ACN just serve as a starting point.

We also recognize that people often do not need to generate a complete UML class diagram. Instead, only the most complex and poorly-understood parts are modeled and analyzed. The effectiveness of using our technique to infer useful information from a partial model is not evaluated yet. Seeking to evaluate the technique using real UML models and a real system is our major future work.

Similar to our UML transformation evaluation, our evaluation on the design rule hierarchy algorithm is still preliminary. We only evaluated the feasibility, but not the effectiveness of this technique in terms of guiding software development and task assignment.

7. Related Work

Organization of software as a layered, hierarchical structure been advocated for many years. Dijkstra [11] proposed this idea with the restriction that layers that could only communicate with adjacent layers. Today, the layered style is popular in software architecture [7]. A major between our layers and these, is that the modules in these architectures are often decided based on classes, or other programmatic components whereas the modules in our approach are independent task assignments. Parnas supported the design of systems with hierarchical structures and the definition of modules as task assignments [26]. He proposed the “uses” hierarchy [27] for organizing programs. The “assumes” relation that define our hierarchy is similar to the “uses” relation but our tasks may span multiple layers of the “uses” hierarchy.

The idea of software clustering is also not new. Many existing approaches can cluster modules and layers from source code [2, 21, 24, 28] but our approach differs in intent from these. Rather, our approach determines modules and layers at design level.

Related to our design rule hierarchy, researchers have proposed methods for task scheduling but the identification of tasks and their dependencies is often left as a prerequisite for these approaches. For example, Jalote and Jain [19] presented an interesting scheduling algorithm that considers task dependencies, available skills, and resource constraints but their approach expects a task graph as an input. Our approach complements theirs in that we can derive the task graph that can be used as the input to their algorithm, and use their approach to elaborate on our hierarchy’s task assignments while considering other issues such as resource constraints.

Previous work on the formalization of UML is not difficult to find. For example, Evans et al [14] formalize UML with Z notation [30], Baresi and Pezzè [5] describe a formalization using high-level Petri nets [16], and Anastasakis et al [1] convert UML to Alloy [18]. To the best of the authors’ knowledge, there are no existing formalizations of UML using analytical decision models. By using DSMs and ACNs, the designers who are familiar with UML models can leverage new modularity and evolution techniques, such as Baldwin and Clark’s net option value analysis [4], and design modularity testing [9].

Our current implementation of our UML transformation tool takes an approach similar to the graph transformations discussed in Baresi and Pezzè [6]. They describe an approach that uses rules, similar to context-free grammars, to parse a concrete graph syntax into an abstract syntax that contains the operational semantics of the diagram. Similarly, our tool uses rules to search a concrete representation of UML (e.g. XMI [25]) and first transforms it to a common intermediate representation before generating the ACN.

8. Conclusion

Despite the promising utility the design rule theory and its supporting analytical decision models, DSM and ACN, have displayed, the theory and the models present sharp learning curves for software designers who are normally trained for UML modeling. Identifying independent modules, defined as independent task assignment in the theory, is also difficult in large-scale systems. We contributed an approach to automatically transform UML class diagrams into ACNs, addressing the first problem, and a design rule hierarchy clustering algorithm to reveal independent task modules and how design tasks can be maximally parallelized. We evaluated our methods using both canonical and widely used examples, as well as a large, open-source

project, and obtained positive and promising results.

References

- [1] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A challenging model transformation. In *Proc of the 10th MODELS*, pages 436–450, Sept. 2007.
- [2] B. Andreopoulos, A. An, V. Tzerpos, and X. Wang. Multiple layer clusterings of large soft systems. In *Proc of the 12th WCRE*, pages 79–88, Nov. 2005.
- [3] Apache Soft Foundation. Apache ant project. <http://ant.apache.org/>.
- [4] C. Y. Baldwin and K. B. Clark. *Design Rules, Vol. 1: The Power of Modularity*. MIT Press, 2000.
- [5] L. Baresi and M. Pezzè. *On Formalizing UML with High-Level Petri Nets*, volume 2001 of *LNCS*, pages 271–300. Springer, 2001.
- [6] L. Baresi and M. Pezzè. *From Graph Transformation to Software Engineering and Back*, volume 3393 of *LNCS*, pages 24–37. Springer, 2005.
- [7] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 2nd edition, 2003.
- [8] Y. Cai. *Modularity in Design: Formal Modeling and Automated Analysis*. PhD thesis, Univ of Virginia, Aug. 2006.
- [9] Y. Cai, S. Huynh, and T. Xie. A framework and tools support for testing modularity of software design. In *Proc of the 22nd ASE*, Nov. 2007.
- [10] Y. Cai and K. Sullivan. Modularity analysis of logical design models. In *Proc of the 21st ASE*, Sept. 2006.
- [11] E. W. Dijkstra. The structure of the “THE”-multiprogramming system. *CACM*, 11(5):341–346, May 1968.
- [12] A. Egyed, W. Shen, and K. Wang. Maintain life perspectives during the refinement of UML class structures. In *Proc of the 8th FASE*, Apr. 2005.
- [13] S. D. Eppinger. Model-based approaches to managing concurrent engineering. *Journal of Engr Design*, 2(4):283–290, 1991.
- [14] A. Evans, R. France, K. Lano, and B. Rumpe. The UML as a formal modeling notation. *Computer Standards and Interfaces*, 19(17):325–334, 1998.
- [15] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Nov. 1994.
- [16] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezzè. A unified high-level petri net formalism for time-critical systems. *TSE*, 17(2):160–172, Feb. 1991.
- [17] S. Huynh, Y. Cai, and W. Shen. Automatic transformation of UML models into analytical decision models. Technical Report DU-CS-08-01, Drexel Univ, Apr. 2008.
- [18] D. Jackson. Alloy: A lightweight object modeling notation. *TOSEM*, 11(2):256–290, 2002.
- [19] P. Jalote and G. Jain. Assigning tasks in a 24-hour soft development model. In *Proc of the 11th APSEC*, pages 309–315, Dec. 2004.
- [20] M. J. LaMantia, Y. Cai, A. D. MacCormack, and J. Rusnak. Analyzing the evolution of large software systems using design structure matrices and design rule theory. In *Proc of the 7th WICSA*, pages 83–92, Feb. 2008.
- [21] Lattix Inc. The lattix approach, 2004.
- [22] C. V. Lopes and S. K. Bajracharya. An analysis of modularity in aspect-oriented design. In *Proc of the 4th AOSD*, pages 15–26, Mar. 2005.
- [23] 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*, 52(7), July 2006.
- [24] B. S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *TSE*, 32(3):193–208, 2006.
- [25] OMG. XML metadata interchange version 2.1, Dec. 2007.
- [26] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *CACM*, 15(12):1053–8, Dec. 1972.
- [27] D. L. Parnas. Designing software for ease of extension and contraction. *TSE*, 5(2):128–138, Mar. 1979.
- [28] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *Proc of the 20th OOPSLA*, Oct. 2005.
- [29] K. Sethi, Y. Cai, S. Huynh, A. Garcia, and C. Sant’Anna. Assessing design modularity and stability using analytical decision models. Technical Report DU-CS-08-03, Drexel Univ, Sept. 2008.
- [30] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 2nd edition, 1992.
- [31] D. V. Steward. The design structure system: A method for managing the design of complex systems. *IEEE Trans on Engr Management*, 28(3):71–84, 1981.
- [32] K. Sullivan, Y. Cai, B. Hallen, and W. G. Griswold. The structure and value of modularity in design. *ACM SIGSOFT Software Engr Notes*, 26(5):99–108, Sept. 2001.